Aztec C65 for the Apple II

USER MANUAL

RELEASE 1.05

08/01/83

INTRODUCTION

The AZTEC C system consists of a comprehensive set of tools for
producing systems of applications software using the C
programming language. The basic system consists of a compiler,
relocating assembler, and linkage editor. The AZTEC C software
system runs on any Apple II system with 48K of memory, a 16K
memory card, and two disk drives. There are no special terminal
requirements.

The major components of the AZTEC C software development system
are the SHELL command parser, the Aztec C native code compiler,
the AZTEC C pseudo-code compiler, the AZTEC relocating pseudo-
code assembler, the AZTEC relocating 6502 assembler, and the
AZTEC linkage editor.

The standard reference for the C language is:

Brian W. Kernighan and Dennis M. Ritchie, The C Programming
Language. Prentice-Hall Inc., 1978, (Englewood Cliffs, N.J.)

Dennis Ritchie originally designed C for the UNIX project. The
above text besides providing the standard definition and
reference for the C language is an excellent tutorial. AZTEC C
can be conveniently used in conjunction with the K & R text for
learning the C language.

The AZTEC C compilers for the Apple II produce both 6502 assembly
language (native code) and a pseudo-code. The pseudo-code tends
to be 50% smaller than native code and is interpreted at run
time. Both the native and psuedo-code are assembled by the
appropriate assembler to produce relocatable object files.
Pseudo-code and native code object files are freely mixed by the
AZTEC linker to produce hybrid modules that will run both native
and interpreted. AZTEC C is written almost entirely in C and is
implemented using both compilers.

AZTEC C supports all of the C language features except for bit
fields. It is suitable for writing system, utility, or text
processing software. The AZTEC C system consists of compilers
that produce native and pseudo-code assembler source,
relocating assemblers, and a linkage editor plus a comprehensive
set of run time library routines.

The AZTEC relocating assemblers produce relocatable object files
that are combined with other relocatable object files and library
routines by the AZTEC LN linkage editor. The linkage editor will
scan through one or more run time libraries and incorporate any
routines that are referenced by the linked modules.

2

TRADEMARKS

AZTEC C65, AZTEC CCI, AZTEC ASI, AZTEC AS65, and AZTEC LN are trademarks of Manx Software Systems. Apple is a trademark of Apple Computers, Inc. UNIX is a trademark of Bell Laboratories.

Aztec C65 V1.05b Release Document

1. Diskette Contents

The system is shipped on six diskettes (or on three "flippy" type diskettes) which contain the following files:

1) STARTUP

SHELL	SHELL command processor
.PROFILE	automatic startup command file
CONFIG	configuration program
TABSET	program to display and modify current value
LDEV	program to load custom device drivers
SHELLDEV.ARC	source to SHELL device drivers and CONFIG
VEDSRC.ARC	source to VED screen editor
PROGSRC.ARC	source to utility programs
OVERLAY.ARC	source to overlay support routines
UTIL.ARC	source to utility library routines
NATIVE.ARC	source to native code support routines

2) ARCHIVES

ARCH	archive maintenance program
FLTSRC.ARC	source to floating point routines
SYSIO.ARC	source to system I/O routines
SHSYS.ARC	source to special SHELL library routines
STDIOSRC.ARC	source to standard I/O library routines

3) C65

C65	first pass of native code C compiler
C65.2	second pass of native code C compiler
AS65	6502 assembler
VED	screen editor
NM	namelist and size of object file program
CMP	compare two files byte by byte
OD	octal/hex dump program
STDIO.H	standard I/O header file
KBCTL.H	ioctl() header file
MATH.H	header file for transcendental functions
ERRNO.H	header file for errors

4) CCI

CCI	pseudo-code C compiler
ASI	pseudo-code assembler
VED	screen editor
NM	namelist and size of object file program
CMP	compare two files byte by byte
OD	octal/hex dump program
STDIO.H	standard I/O header file
KBCTL.H	ioctl() header file
MATH.H	header file for transcendental functions
ERRNO.H	header file for errors

5) LIBINT
    MKLIB   library maintenance program
    SHINT.LIB   pseudo-code version of SHELL library
    FLTINT.LIB   pseudo-code version of FLOAT library
    SA65.LIB   native code version of STAND ALONE library

6) LIB65
    LN   linker
    SH65.LIB   native code version of SHELL library
    FLT65.LIB   native code version of FLOAT library
    SAINT.LIB   pseudo-code version of STAND ALONE library

The source archives to the libraries are organized as follows:

FLOAT library - FLTSRC.ARC

SHELL library - SHSYS.ARC UTIL.ARC STDIOSRC.ARC NATIVE.ARC

STAND ALONE   - SYSIO.ARC UTIL.ARC STDIOSRC.ARC NATIVE.ARC

2. Changes from 1.05a

The new SHELL uses a different approach to device drivers entirely which provides support for a wider variety of printers and 80 column cards. However, the old drivers are still valid.

A number of bugs in the C65 compiler were fixed, and a few bugs in the library were fixed as well (gets/fgets in particular).

The current execution drive is now accessible through a "-1" mode in open() and an "rx" mode in fopen(). This mode is now used by both compilers when looking for "#include" files and by the linker when looking for both libraries and object files.

The memory allocation routines were modified to allow for allocation of space which cannot be freed.

There is a system-wide idea of the size of a tab which can be modified with the TABSET program.

Overlays are now supported.

Jim Goodnow II

August 8, 1983

APPENDICES

Tutorial Introduction

to

Aztec C65

SECTION 1 - TUTORIAL INTRODUCTION TO AZTEC C65

1.1 Getting Started

Congratulations on purchasing the Aztec C65 compiler from Manx Software Systems. This part of the manual contains sections on installing and configuring the SHELL command processor to your Apple. It then proceeds step by step through the creation, compiling, linking and running of a test program. On the way, it will introduce important parts of the SHELL which give a very UNIX-like environment on a small machine. The remainder of the manual is more of a reference guide and provides more detailed information on the individual pieces and procedures.

The Aztec C65 system is shipped on either three reversible or six single sided diskettes. These diskettes should be copied to six other single sided diskettes before being used. These diskettes have been initialized with a special program which allows files to be stored on tracks 1 and 2 which are normally reserved for DOS. Therefore, the best way to copy them is to use the COPY or COPYA program which is supplied on the DOS 3.3 master. The originals should then be stored in a safe place in case they are needed again.

To boot the SHELL, first boot DOS 3.3 from the DOS 3.3 system master supplied with your Apple. **The disks supplied by Manx cannot be booted.** Then, insert the disk labelled STARTUP into drive 1 and type:

BRUN SHELL

This is a binary program which contains the new command processor, the pseudo-code interpreter, and part of the library. It will automatically move itself to the appropriate places in memory. For more information, consult the memory map in the SHELL section of this manual.

The SHELL will display the message:

APPLE][SHELL 2.X

COPYRIGHT (C) 1983
BY MANX SOFTWARE SYSTEMS

on the screen and the prompt:

-?

Following the prompt should be a solid cursor. At this point, the SHELL is up and running, and assumes that the Apple it is running on is a normal bare bones Apple II.

In this configuration, the ESC key acts as a caps lock/unlock key, lower case is displayed as normal text, and upper case is displayed as inverse video. Try typing some characters. They should appear as normal video characters. Now,

press the ESC key. The first thing you should notice, is that the cursor is now flashing. This signifies that you are in CAPS LOCK mode. Try typing some characters now. They should appear as inverse video characters, which indicates that they are upper case.

If you are using a keyboard with full upper and lower case capability, such as the Apple IIe, you will need to type the characters as upper case. This is necessary since the SHELL is delivered configured for a basic Apple II. We will discuss how to take advantage of additional capability shortly. On the IIe, simply make sure that the CAPS LOCK key is engaged.

Type ^X to cancel the line that you typed. (Note that control characters will sometimes be displayed in this manual as a caret followed by the appropriate character.) There are a number of other control characters which have special meaning. The full list is discussed in the SHELL reference section on console I/O.

The important ones are:

> ^H (also the left arrow key) which is used to backspace
>   over the last character typed.
> ^X to cancel the current line of input.
> ^S to stop and restart output to the screen.
> ^C to abort a program and return control to the SHELL.

Under the SHELL, the command to catalog the disk is "ls". Try typing it now followed by a return to see the files on the STARTUP diskette. The SHELL normally assumes that commands are typed in lower case. If you typed "LS", the SHELL tried to find a file with the name "LS" to run, and gave an error message when it didn't find it. Hit the ESC key to get out of CAPS LOCK and try it again. The "ls" command is "built-in" to the SHELL. A full list of the built-in commands can be found in the SHELL reference section.

1.2 Configuring the SHELL

Up to this point, the SHELL has ignored any peripherals or options which you might have added to your machine. To make use of these features, the SHELL must be configured to the exact system which you are using. This is done by using the CONFIG program which is also on the STARTUP disk. To run the CONFIG program, simply type:

config

followed by a return. Note that unlike DOS, you don't need to type RUN or BRUN to execute programs. Simply the name of a file will cause it to be loaded and executed.

When the CONFIG program has been loaded, it will display a startup message and ask a series of questions about the machine

you are using and the peripherals installed. Most questions can
be answered with a simple 'y' or 'n'. A more detailed discussion
of the CONFIG program and the meaning of some of the questions
can be found in the CONFIG reference section.

At one point in the program, it will ask if you are using an
80-column video card. If you answer yes, it will ask about
specific cards that it has tables for. If the card you are using
is not in this list, you must provide information from the card's
manual. For the purpose of this introduction, you may wish to
cancel the CONFIG program and perform the configuration later
after reading the CONFIG reference section. In the meantime, the
default configuration should suffice till then.

When the configuration is finished, the program will ask if
you wish to store that configuration. If you answer 'n', only the
current memory version of the SHELL will be altered. Answering
'y' will write the configuration information into the SHELL
binary file so that the next time the SHELL is booted, it will
already be configured.

1.3 Two Drive Environment

One of the nice features of the SHELL is it's use of two
drive disk systems. To illustrate this, insert a DOS initialized
diskette into drive two. To catalog drive 2, type:

   ls d2

This is different from the DOS way of doing things in two
ways. The command name "ls" must be separated from its argument
by at least one space, not a comma. The second thing that is
different, is that this command does not make drive two the
active drive. Typing the "ls" command by itself will still give
the catalog for drive one. To change the active drive, the "cd"
command must be used. Type:

   cd d2

to change the active drive from drive one to drive two. Now drive
two will be the active drive until another "cd" command is given,
or the system is rebooted.

The other nice feature for multi-drive systems is the
concept of an execution drive. For example, try typing:

   config

Note that the drive light on the active drive will go on as the
SHELL tries to find the program. Assuming that you don't have a
program named CONFIG on the scratch disk, the SHELL will then
automatically check the current execution drive. In this case the
current execution drive will be drive one, and the CONFIG program
will be loaded. The current execution drive can be changed by
using the "ce" command in a manner similar to the "cd" command.

For the rest of this introduction, it will be assumed that the current execution drive is drive one, and that the current data drive is drive two. After cancelling out of the CONFIG program by typing ^C, type the following just to be sure:

      ce d1
      cd d2

Then replace the STARTUP disk in drive one with the disk labelled C65.

1.4 Creating the Program

The C65 disk contains the 6502 C compiler, the 6502 assembler, and several utility programs. In the following paragraphs, we will use the VED screen editor to create a test program which we will then compile, assemble and link with a library. The result will be a file which we can then execute. We will also make use of some of the other utilities as well.

The program which we will write gives a useful demonstration of how arguments passed to a program are accessed by the program. The following is a listing of the program:

```
main(argc, argv)
int argc;
char *argv[];
{
    register int i = 1;

    printf("Program <%s> has %d arguments\n", argv[0], argc-1);
    while (--argc) {
        printf("Arg %d = <%s>\n", i, argv[i]);
        i++;
    }
}
```

As can be seen, the program prints it's name, which is the first argument, and the number of arguments. Since the number of arguments includes the program name, argc-1 is used as the number of real arguments. Then, each argument is listed on a separate line.

The first step is to create the source program using the VED screen editor. Type:

      ved args.c

VED will be loaded from the current execution drive, and will try to find "args.c" on the disk. When it doesn't find it, it will

say so and will start with an empty document. Note that the screen should look like:

```
"args.c" line 1 of 1

-
-
-
```

The cursor should be on the second line, and a single '-' on all the remaining lines. The '-' indicates that the line is after the end of the file. If the screen does not look this way, there is something wrong with the way that your SHELL is configured. Refer to the CONFIG reference section before proceeding further.

VED has two modes, command and insert. Normally, VED is in command mode. For a list of most of the commands available, try typing a question mark without a return. The screen should clear, and the list should appear. Pressing the return key should repaint the screen with the document being edited. To enter insert mode, simply press the 'i' key. On the status line, the <INSERT> mode indicator should appear. This will always be there when in insert mode.

At this point, type in the test program, using the left arrow key to correct any mistakes. The indentation in the program is produced by using a tab character. The tab character width is defined by the SHELL and defaults to four. It can be changed using the TABSET program discussed in the PROGRAMS section of this reference manual.

If you are using a standard Apple II keyboard, you will need help to produce some of the characters. To get the '{', type ^A. To get the '[', first press the ESC key to go to CAPS LOCK mode and then type ^A. If you have installed the SWSKM (single wire shift key mod), and configured the SHELL for it, then use shift ^A to get the '['.

The following table lists the other mappings you will need. The capitalized control characters must be typed with the CAPS LOCK on or the shift key down if the SWSKM is installed.

^a	->	{
^A	->	[
^E	->	\
^I	->	tab (the right arrow key on Apple II's may be used as well)
^r	->	}
^R	->	]

VED expects an ESC character to end the insert mode. If the ESC key is being used as a CAPS LOCK key, the ^Q key will produce an ESC character instead. Once out of insert mode, the cursor can be moved around using the space bar to move right and the left arrow to move left. To move a number of characters to the right

or left, type the number of characters to skip followed by the space or backspace. To move to the beginning of the next line, use the return key. Similarly, use the '-' key to move to the beginning of the previous line. Characters can be deleted by placing the cursor on the character and pressing the 'x' key. Characters can be inserted by placing the cursor at the insertion point and pressing 'i' to enter insert mode.

When the program has been entered and corrected, type ":w" followed by a return. This will write the document to the file we originally tried to edit, "args.c". To write the document to a different file, simply type ":w file.c" followed by a return. When the file has been written, exit the editor by typing ":q" followed by return. If you try to exit without writing the file, VED will display the message:

file modified - use q! to override

This message will appear whenever you try to exit VED after making a change without writing the file out. To exit without saving the changes made, type ":q!" followed by return.

1.5 More SHELL Goodies

At this point, the SHELL prompt should be back. To examine the file you created, you may either use VED again, or type:

cat args.c

to display the file on the screen. This uses the built-in SHELL command, "cat", which opens it's arguments one by one and copies them to the standard output.

If you have a printer card installed and configured correctly, you can print the file with the following command:

cat args.c > pr:

This introduces another feature of the SHELL, I/O redirection. Under the SHELL, when a program is invoked, it has three pre-opened channels of communication. These are usually referred to as the standard input, output and error. Normally, the standard input channel is connected to the keyboard, while the standard output and error channels are both connected to the screen. However, by using the special characters '<' and '>', the standard input and output can be "redirected" to other devices.

Thus in the above examples, the "cat" command opens the file specified by the argument and reads the contents of that file and writes them to the standard output. In the first case, that was the screen. By using the "> pr:" in the second example, the SHELL switched the standard output to "pr:" which is the name of the printer device. The name of both the keyboard and screen is "kb:". We will say more about I/O redirection later.

1.6 C65 and CCI, The Speed Versus Size Dilemma

Now that we have our C source program, the time has come to compile it. The Aztec C65 system actually comes with two C compilers. The first compiler, C65, produces 6502 machine code, while the second compiler, CCI, produces a pseudo-code that must be interpreted. Because of the architecture of the 6502 microprocessor, there are advantages to both.

The 6502 microprocessor is completely restricted to dealing with single bytes at a time. Addresses and numbers larger than 256, on the other hand, are two bytes in size. As a result, the 6502 machine code generated by C65 tends to be larger than programs produced for machines which have better facilities for handling 16-bit quantities. As an alternative, the pseudo-code C compiler, CCI, produces machine language for a theoretical machine with 8, 16 and 32 bit capabilities. This machine language is interpreted by an assembly language program that is about 3000 bytes in size.

The effects of using CCI, are twofold. First, since one instruction can manipulate a 16 or 32 bit quantity, the size of the compiled program is generally more than fifty percent smaller than the same program compiled with C65. However, interpreting the pseudo-code incurs an overhead which causes the execution speed to be anywhere from five to twenty times slower.

The dilemma appears obvious: speed versus size. For most applications, hopefully, the resolution is obvious. If the program is small, there should be no problem using C65. If the program is large and the speed of execution not critical, use CCI. If the program is large and execution speed important, there are at least three solutions.

First, code extremely time critical parts of the program directly in 6502 assembly language. This is typically necessary in applications such as real-time graphics, where the overall program is written in a higher level language, but the extremely time-critical portions are written in assembly. A second approach, similar to the first, is to compile just the time critical routines with C65 and the remaining routines with CCI. Both compilers and assemblers have been designed so that the object modules produced by each may be combined together into one binary program.

A third possibility is to use the overlay facility provided with this system. Overlays allow portions of a program to be loaded from a disk when they are needed, and then to be "overlaid" with other portions. Using this technique, the size of a program need only be limited by the size of the disk you are using. Finally, any combination of the above methods may be used to achieve a satisfactory balance of size and execution speed.

1.7 Compiling and Assembling

The examples and discussion which follow are restricted to C65, but basically apply to CCI as well.

The simplest way to use C65 is to type:

c65 args.c

The compiler will be loaded from the execution drive and will display the version number and the copyright message. It then translates the source file into 6502 assembly language. The assembly language is placed in a file called "$TMP.$$$" which will be deleted later by the assembler. After the compiler finishes, the 6502 assembler, AS65, is automatically loaded. AS65 assembles the assembly language in "$TMP.$$$" and places its output in a file called "args.rel". The type of the ".rel" file is 'R' indicating that it is a relocatable object file. When the assembler finishes, it deletes the temporary file, "$TMP.$$$". At this point the compile is finished.

While the source is being compiled, if any errors are detected, the line containing the error will be displayed, along with the line number and the error number. Refer to the error summary in the appendix to translate the error number. If there is an error, use VED to edit the file. To move the cursor to the line with the error, type the line number followed by a 'g'. Correct the error, write the file, and recompile it.

If you wish to compile without assembling, then typing:

c65 -a args.c

will compile the program and produce an assembly language text file called "args.asm". This file may be edited or printed as desired. When this option is used, the assembler is not automatically executed.

To produce a relocatable object file from "args.asm", type:

as65 args.asm

AS65 will place its output in a file called "args.rel". Note that in this case, AS65 will not delete "args.asm" when it is finished.

1.8 A Few Utilities

The relocatable object file produced by both assemblers is in a special binary format. If you "cat" the file to the screen, the result will be visual garbage. Instead, to look at the contents of a non-text file, there is a utility program called

OD. This is not a built-in command, and must be loaded from the disk. To execute the program, type:

od args.rel

The program will open the file args.rel and display in hex the value of each byte in the file. If the byte is also a displayable character, it will be displayed at the right of the hex values. Non-displayable characters will be displayed as a period. The display can be temporarily stopped and restarted by using the ^S key. The program can be aborted by typing ^C. OD can be used to dump the contents of any file, text, binary, basic, or relocatable object. Try it on "args.c".

A second utility, NM, works only with relocatable object modules. This utility performs two functions. First, it can display the size of the code and the data which is contained in an object file. This is useful since the physical size of an object file does not directly reflect the size of the code and data which will be produced when it is converted to absolute binary form. To see the size of the code produced by the "args.c" program, type:

nm -s args.rel

The result should be about 210 bytes if you used C65, and 96 bytes if you used CCI.

The second function of NM is to display the names and offsets, if known, of all labels defined in a module. This is mostly useful when building libraries. It is possible to determine what labels are defined within this module and which are yet to be defined. The various options for the output can be found in the PROGRAMS reference section. Typing:

nm args.rel

will show that the "main_" function is defined in this file, and that several functions are undefined, including "printf_".

1.9 Linking with the Library

Both assemblers translate assembly language into a format called relocatable object format. This format is designed to allow the program module to be converted into absolute data which will be loaded and run at a specific address in memory. This becomes particularly important when the final program consists of several modules compiled and assembled separately. As will be seen, this is true of almost all programs.

For example, assume that a program consists of two modules, "main.rel" and "subs.rel". Assume, also, that "subs" contains several functions to be called from "main". Since the two modules are compiled and assembled separately, there is no way for "main"

to know where "subs" is going to be in memory. Even if "main" did
know the address of the beginning of the "subs" module, it has no
way of knowing the size of each function in that module.

It is possible that one could give all the information
needed when compiling and assembling "main" to directly produce a
binary image. This is only practical if the amount of information
needed is quite small. However, most C programs make use of a
number of functions supplied with the compiler. These functions
are usually kept in individual modules so that functions not used
by the program are not included.

The number of these functions make it totally impractical to
produce any kind of direct binary output. The solution is the
relocatable object format and a program to link object modules
together, the Aztec linker, LN.

LN combines any number of object modules together and
produces a binary file in the standard Apple DOS "BRUN" format.
LN will also indicate if anything is missing. For this example
replace the C65 disk with the LIB65 disk and type:

ln args.rel

In this case, LN will attempt to produce a binary file from
"args.rel". However, since the "args" program makes reference to
several functions which are not defined in the "args" module, the
linker will give error messages to that effect.

Supplied with the Aztec C65 system, is a large set of
subroutines which perform many different functions. A large
percentage of these routines are used to perform input and output
operations, since the C language has no inherent mechanisms for
doing I/O. A complete list of these functions and a description
of each can be found in the LIBRARIES section of this reference
manual.

To simplify the process of selecting the correct routines to
be linked with a particular program, it is possible to combine a
number of routines into a single file, called a library. The
format of a library is designed so that individual modules can be
read from it without reading all the modules. In addition, the
linker, LN, will search a library and only use those modules
which satisfy references made in other modules that it has
processed.

Thus, to correctly link the "args" program, type:

ln args.rel sh65.lib

In this case, the linker will read the "args.rel" file and make a
list of all undefined symbols. Then, it will check the library
(note that LN looks for modules or libraries on both the data and
execution drives automatically) for any modules which contain the
proper symbol. If it finds one, it will read that module from the

To save the output of the "args" program in a file, we can
use the I/O redirection capability of the SHELL. The printf()
routine that we used in "args", sends its data to the standard
output which can be redirected, as in:

    args one two three > args.out
    cat args.out

The first line calls "args" with three arguments. The '>'
and all following information is directed to the SHELL and is not
passed to the program. The file "args.out" now contains the
output that would have gone to the screen. I/O redirection can be
used to redirect I/O to or from disk files, or the devices "kb:"
and "pr:".

1.11 More Choices

There are basically two libraries supplied with the Aztec C
system. One contains the transcendental math functions and the
floating point emulation routines. The second contains all the
other routines. When linking, the FLOAT library need only be
specified if floating point is used somewhere within one of the
modules. If floating point has been used, and the program is
linked without the FLOAT library, there will probably be a number
of unresolved references. In particular, the symbol, ".fltused",
indicates that floating point was used at some point. This is an
example of a case where the NM program could be used to determine
which modules declared ".fltused" as undefined.

When linking with the FLOAT library, it should be placed
before the regular library in the argument list. For example:

    ln -o flargs args.rel flt65.lib sh65.lib

will create a binary program called "flargs" which contains the
floating point emulation routines.

Although there are only two basic libraries, there are a
number of different flavors of each. The FLOAT library comes in
only two flavors, FLT65.LIB and FLTINT.LIB. Both libraries
contain the same functions, but all the C language routines in
FLT65.LIB have been compiled with C65, while those in FLTINT.LIB
with CCI.

The regular library also comes in a C65 version and a CCI
version. However, there is another distinction as well. One
version of the regular library is only useful when creating
programs that will run while the SHELL is in memory. The other
version is designed to allow programs to run directly under Apple
DOS with or without the SHELL. The second version is called the
STAND-ALONE library.

The SHELL libraries are called SH65.LIB and SHINT.LIB which
corespond to the C65 and CCI versions respectively. Likewise, the

STAND-ALONE libraries are called SA65.LIB and SAINT.LIB. All the libraries compiled with C65 are on the disk labelled LIB65, while the disk labelled LIBINT contain the others.

The differences between the STAND-ALONE library and the SHELL library are discussed in the LIBRARIES section of this manual. The FLOAT libraries may be used stand-alone or with the SHELL.

1.12 Going to the Source

The source to most of the library routines, some of the utility programs, and parts of the SHELL, are included with the Aztec C system. These text files are collected together in a set of binary files called archives. Placing the files in archives allows more efficient use of the disk space. Replace the disk in the current execution drive (drive 1) with the disk labelled ARCHIVES and type:

        cp progsrc.arc,d1 progsrc.arc

The "built-in" command, "cp", will copy the file "progsrc.arc" from drive one to the current data drive (drive 2). Now type:

        arch -l -o progsrc.arc

The "-l" option tells the ARCH program to list the names of the files in the archive. The name of the archive is specified by using the "-o" option. ARCH will list the name and size of each file in the archive. To extract one of the files, type:

        arch -x -o progsrc.arc tabset.c

The "-x" option tells ARCH to extract the file names which are passed as arguments. Thus, more than one name may be specified at a time. That is also why the "-o" option is necessary to tell ARCH which argument is the archive itself. If the "-x" argument is specified with no filenames, then all the files in the archive are extracted.

Included with this manual should be a release document which describes the contents of each archive.

1.13 A Few Last SHELL Commands

To remove the archive files from the scratch, type:

        rm progsrc.arc tabset.c

The "rm" command acts like the DOS "DELETE" command, except, that more than one name may be specified at one time. The SHELL "lock" and "unlock" commands work in a similar fashion. One last

command, a clever one, is the "mv" command. This command takes
two arguments, a source and a destination, and moves the source
to the destination. If the source and destination are on the same
disk, "mv" simply renames the file. If the source and destination
are on different disks, "mv" "cp"'s the file from the source to
the destination and then "rm"'s the source.

A full list of all the SHELL "built-in" commands can be
found in the SHELL section of this manual.

1.14 Where to Go From Here

Well, that about covers the basics. The rest of this manual
is devoted to giving more precise technical information on a
number of different topics. The major sections and their contents
can be summarized as follows:

SHELL	-	commands and features
PROGRAMS	--	options and use of each program
LIBRARIES	-	calling sequence and function
TECH INFO	-	a variety of information

Familiarity with the sections on the SHELL and options to
the programs is highly recommended. In the beginning of the
libraries section, there are several sheets which provide a
summary of the library functions and their arguments. A copy of
these sheets along with a copy of the compiler error codes can be
found as the last pages of this manual and can be used as a handy
reference. The last section, contains a number of different
documents which provide information on a variety of topics,
including overlays, floating point format, ROMable code, device
drivers, stand-alone use and others.

1.15 Complaint Department

Finally, if there are any questions, suggestions,
criticisms, bugs, and especially compliments, feel free to use
the enclosed "PROBLEM REPORT FORM" or just drop a card or send a
letter. We may or may not respond, but we'll certainly read it
and if there's a problem we'll do something about it. Placing a
problem or request in writing increases the probability of its
being remembered exponentially, as I have a terrible memory and
the notes I make on the phone are often illegible. As promised in
the past, the system and compilers have continued to improve and
Manx is committed to the future as well.

The Shell

SECTION 2 - SHELL

2.1 Introduction

The Apple SHELL is a program written mostly in C which replaces and assumes the functions of the Apple DOS 3.3 Basic command interpreter. The commands recognized by the SHELL are similar in function to the commands recognized by the DOS command interpreter. However, they provide some additional functionality as well as providing a friendlier C environment. This environment is UNIX-like in a number of ways.

It is important to note that the Apple SHELL is designed primarily for the execution of binary programs produced by the Aztec C compiler system. Execution of Integer or Applesoft Basic programs is not possible while using the SHELL. Binary programs which have been produced without using the Aztec C compiler system may or may not run. They will have to be tested on a per program basis by the user.

There is a second advantage to using the Apple SHELL. The SHELL uses a number of the basic library routines. Rather than duplicate them in application programs running under the SHELL, these programs can access the library routines already present in the SHELL through a special vector. This saves space in each application program. These programs may also be linked with the stand-alone library to run without the SHELL being present.

2.2 Booting

To run the Apple SHELL, you must first boot DOS 3.3 from a DOS 3.3 initialized disk, the way you normally would. **NOTE: Diskettes supplied with the ACCS can not be booted.** Now, insert the disk labeled STARTUP into drive number 1. The Apple must contain a 16K ram-card of some sort. Type BRUN SHELL. The SHELL resides in the ram-card and allows more space for application programs. The SHELL overwrites the Basic command parser of DOS 3.3. See the memory map for more details.

At this point, the screen should clear and display the message:

APPLE][SHELL 2.X

COPYRIGHT (C) 1983
BY MANX SOFTWARE SYSTEMS

at the top of the screen with the prompt "-?" on the next line. There should also be a solid cursor following the prompt. Try typing "ls" followed by a return. This should give the normal DOS catalog of the disk from which the SHELL was booted.

Because the SHELL removes the BASIC command parser part of DOS 3.3, any modified DOS's may not work properly with the SHELL. Usually DOS is modified to enhance it's capabilities/execution speed, or to incorporate a different device such as a hard or eight-inch disk drive. The best thing is to try it and see. If

strange things happen, try the same thing with normal DOS 3.3 and see if the strange things stop happening.

Unfortunately, the DOS file manager and the Basic command parser are not completely independent. There are at least two places in the DOS file manager, where this occurs. When the SHELL boots, the boot program verifies that the version of DOS in memory corresponds exactly to the area which needs to be patched. If the area matches, the area is patched, otherwise the message:

INCORRECT DOS VERSION

is displayed, and the SHELL is not booted.

2.3   Default Configuration

When the SHELL is booted for the first time, it knows about four devices, a keyboard, screen, printer and disk. The SHELL contains its own routines for doing input and output. These routines use a table of configuration information to allow for the different flavors of Apple computers in use today. As shipped, this table is set up to work with a basic Apple without any enhancements.

Changing the table is made fairly easy through the CONFIG program which is described in the configuration section of this part of the manual. However, until the CONFIG program is used, the following information on the default configuration may prove useful.

First, since the SHELL makes extensive use of lower case for its commands, but the normal Apple keyboard produces only upper case, all upper case alphabetic characters are translated to lower case. Lower case characters produced by the IIe and enhanced keyboards will cause indeterminate results. Thus, until properly configured, these keyboards should be used with the CAPS LOCK key engaged. For more information on the operation of the keyboard mapping, consult the console I/O section.

Second, since the normal Apple screen can only display upper case, the SHELL maps upper case characters to inverse, and displays lower case as non-inverse. In addition, certain characters not found in the normal Apple character rom are displayed using inverse. For example, '{' is displayed as an inverse '('.

The printer card is assumed to be in slot one and to work using the normal Apple PR# convention. The card is initialized with the string:

^I^Y^Y255N

Characters sent to the printer all have the high bit on and carriage returns are followed by line feeds.

2.4 Console I/O

Under most configurations, the Apple keyboard is read directly by the SHELL keyboard routine. The SHELL keyboard input routine can perform several functions. The primary function is to allow the user to enter both upper and lower case from an upper case only keyboard. This is done either through the use of the SHIFT key or a CAPS LOCK key.

In order to make use of the SHIFT key, a wire must be connected between the shift key (or the second pin from the right on the keyboard to piggy-back board connector) and pin one of chip H14 (which is the same as pin four of the game connector). If there is any question as to how to make the connection, we recommend that the user have their dealer make this modification.

If the single wire shift key mod (SWSKM) is not implemented, the key which performs the CAPS LOCK function is the ESC key. Each time it is pressed, it switches from the current case to the opposite. When the normal Apple screen is in use, the current case can be determined by the type of cursor on the screen. If the cursor is solid, characters typed will be in lower case. If the cursor is blinking, characters typed are in upper case.

There are several characters which cannot normally be accessed from the apple keyboard. These characters and the keys which must be pressed to obtain them are shown in the following table (where the LOWER and UPPER signify the state of the SHIFT KEY or CAPS LOCK):

PRESS:	TO GET (LOWER):		TO GET (UPPER):	
control P	`	0x60	@	0x40
control A	{	0x7B	[	0x5B
control E	|	0x7C	\	0x5C
control R	}	0x7D	]	0x5D
control N	~	0x7E	^	0x5E
control C	DEL	0x7F	_	0x5F

When the keyboard mapping is enabled, two other characters are also mapped to other values.

control Q sends a real ESC character (only if CAPS LOCK is enabled).

control U the right facing arrow at the far right of the Apple keyboard produces the control U code which is mapped into the code for a tab (control I).

If the keyboard mapping is not enabled, then the control characters tabled above take on their normal values. The following control characters, however, will have special meaning to the SHELL unless specifically disabled (see Device Ioctl in Technical Information section).

control C	when printing to screen or waiting for input, causes the program to exit.
control H	backspaces over the character to the left of the cursor.
delete	same as control H.
control S	causes the screen to pause until the next control S is hit.
control X	deletes the current input line.
return	is converted to a newline character (line feed).
control D	EOF from keyboard.

As the characters are received by the keyboard routine, they are echoed through the screen routine. The screen routine's primary function parallels that of the keyboard routine, the display of upper and lower case on the Apple screen. This is done by using inverse video to designate upper case and special characters not in the standard Apple rom. If the Apple has been enhanced to display upper/lower case on the screen, this mapping can be disabled. The screen routine also translates carriage returns or line feeds into a carriage/line feed combination.

The keyboard and screen can be accessed by programs through the "KB:" device.

2.5 General Use

The simplest form of a SHELL command is the name of a function followed by a carriage return. A SHELL command may either be one of the built-in utilities or the name of a binary or text file which resides on disk. The following is a list of the built-in functions available with the SHELL. If a file has the same name as one of these functions, the SHELL will not execute that file, but will execute the built-in function instead.

boot	cp	mv
bye	load	rm
call	lock	run
cat	ls	save
cd/ce	maxfiles	unlock

These commands are all specified using lower case. A complete description of each command can be found in the Commands section.

Binary programs which are normally run using the DOS 'BRUN' command can be loaded and executed by simply typing the name of the file followed by a carriage return. The first two words of binary files which contain executable programs contain the load address and length in bytes of the memory image. These are used to load the program into memory. The SHELL 'load' command can be used to load the image into a different section of memory much the same as the DOS 'BLOAD' command.

Text files containing a series of SHELL command lines can be
executed by simply typing the name of the text file followed by a
carriage return. All SHELL input is then taken from that file
until the end is reached. For more information see the Batch
Facilities section.

Some built-in SHELL utilities as well as binary programs
produced using the Aztec C compiler system require or allow
arguments to be specified when the command is executed. These
arguments are placed on the same line as the command name
separated by spaces. An example of this is the SHELL 'lock'
command. Under Apple DOS, if it is desired to lock several files,
the DOS 'LOCK' command must be given once for each file. To lock
several files using the SHELL, the user would type something
like:

lock test1 test2 test3,d2

to lock files "test1" and "test2" on the current drive and
"test3" on drive two.

Because arguments are separated by spaces, file names
containing spaces must be enclosed in double quotes to enable the
SHELL to distinguish the single name from two names. For example,
to unlock a file called "test prog", the user would type:

unlock "test prog"

Double quotes should also be used around file names used as
commands if the name contains any blanks.

The final feature of the SHELL which will be discussed is
the ability to redirect the standard input and/or output of a
program to a file or a device. Normally the standard input and
output of a program are connected to the keyboard and screen
respectively. The user may redirect either or both of these
connections to a file or a device such as a printer. This is
accomplished by using the special character '<' for input and '>'
for output.

As an example, to place the output of the NM command, which
produces a symbol table from an object file, into a file for
later perusal, type:

nm objfile >listing

The namelist will not be printed to the screen, but to the file
'listing' instead.

The SHELL also pre-opens a second channel to the screen
called the standard error output. This channel cannot be
redirected.

2.6 Built-in Commands

This section describes the commands which are built into the SHELL program itself. Each SHELL command will be listed along with a description of its use and its function. All commands are specified as being lower case. File names may be typed with either upper or lower case letters, however they will all be mapped to upper case for compatibility with Apple DOS. File names may contain blanks, but to distinguish arguments from the parts of the file name, the entire name must be enclosed within double quotes.

In the following discussions, the concept of current data drive and current execution drive are used. Under DOS, the last drive accessed is considered the current drive. Under the SHELL, the current data slot, drive and volume must be explicitly changed by the user using the "cd" command. At any point where an optional slot, drive or volume parameter may be given, if any are not specified, they will default to the current data value respectively. The examples given for specific commands should clarify this point.

In general, all arguments to SHELL commands and to utility programs are separated by blanks. Arguments in square brackets are optional and most commands allow more than one file name per command line. In the following descriptions, any reference to a file name is assumed to include the optional slot, drive and volume parameters.

2.6.1 boot

boot n

Does a jump to address $Cn00. (Usually to reboot.)

Example:

boot 6

Causes the floppy disk to reboot.

2.6.2 bye

bye

Does a jump to the Apple machine language monitor at location $FF65. Reentry to the SHELL is through $3D0 or by hitting RESET on systems with the autostart rom.

2.6.3 call

call addr

Performs a "jsr" to the address given. If addr is preceded by a '$', it is interpreted as hex, otherwise as decimal.

Examples:

call $800
call -151

The first example does a "jsr" to hex 800, while the second calls the monitor.

2.6.4 cat

cat [file1] [file2] ...

Concatenates the named files to the standard output. If no files are specified, input is taken from the standard input. This is the quickest and easiest way of looking at a text file.

Examples:

cat test1 test2,d1
cat test1 test2,d1 > test3
cat > pr:
cat kb: > pr:

The first example displays "test1" from the current data drive on the screen immediately followed by the file "test2" located on drive one. The second example creates a new file called "test3" containing the two files "test1" and "test2". The third example reads a character from the standard input and writes it to the device "pr:" which is the printer. The fourth example is equivalent to the third.

2.6.5 cd

cd sn,dn,vn

Change the current data slot, drive and/or volume. Any or all of the three parameters may be changed. Those not specified will remain the same. If a volume number is specified, it will be checked whenever a file is opened. A volume number of zero, however, will match any disk.

Examples:

    cd s6,d1,v0
    cd d2

The first example changes the current data disk to be slot six, drive one, and any volume. The second example changes from whatever the current drive was to drive two. The slot and volume remain the same.

2.6.6 ce

    ce sn,dn,vn

Change the current execution slot, drive and/or volume. Execution parameters are used when loading and running a particular binary program or SHELL file. If the name includes a specific reference to a slot, drive or volume, that parameter is used. If there is no reference as to which device holds the file, the current data disk is searched and if the file is not found there, then the current execution disk is checked. This allows all utility programs to reside on a different disk than the one being actively used.

    ce s6,d2,v0
    ce d1

The first example changes the current execution disk to be slot six, drive two, and any volume. The second example changes from whatever the current drive was to drive one. The slot and volume remain the same.

2.6.7 cp

    cp file1 file2

Copies file1 from the specified device to file2. Note that file2 will be overwritten if it already exists.

Examples:

    cp test oldtest
    cp test,d1 test

The first example makes a copy of "test" on the same disk called "oldtest". The second example assumes that drive one is not the current data drive and copies the file "test" from drive one to the current drive.

2.6.8 load

load file [aN] [lN]

Loads a binary file into memory. If the starting address
and/or length are not specified, they are taken from the first
two words of the file. After loading, the start address and
length are displayed on the screen. These values are remembered
for use in the save and run commands. If N begins with a '$', the
value is interpreted as a hex value otherwise as decimal.

Examples:

> **load tabset**
> A=0800 L=12F2
>
> **load tabset a$2000**
> A=2000 L=12F2

The first example loads the tabset program into memory. The
shell displays the load address and length. The second example
loads the tabset program into memory at address hex 2000.

2.6.9 lock

lock file1 [file2] ...

Lock the file on the specified slot, drive and volume. If
any of slot, drive or volume are not given, they default to the
current data values.

Examples:

> **lock test1**
> **lock test1 test2 test3,d2**

The first example locks file test1 on the current data disk.
Example two locks files test1 and test2 on the current data disk
and locks file test3 on drive two of the current data slot and
volume.

2.6.10 ls

ls [sn,dn,vn] ...

Perform the catalog function on the specified slot, drive
and volume. This command defaults to the current data slot, drive
and volume. If more than one is specified, they will be cataloged
in order. The SHELL will wait for a key to be pressed between
different catalogs. Unfortunately, the output of ls cannot be
redirected.

Examples:

    ls
    ls d1 d2

The first example does a catalog of the current data slot, drive and volume. The second example catalogs drive one and then drive two of the current data slot and volume.

2.6.11 maxfiles

maxfiles n

Allocates n buffers for open files. This command is similar to the DOS 'MAXFILES' command. It specifies the maximum number of disk files which may be open at any one time. When the SHELL is initialized, the value is defaulted to 3.

Example:

    maxfile 4

For an application which will have four disk files open, maxfiles is set to four.

2.6.12 mv

mv [-f] file1 file2

Moves file1 to file2. If the slot, drive and volume of file1 are the same as that of file2, file1 is simply renamed as file2. If they are different, file1 is copied to file2 on the specified device and file1 is deleted. If file2 exists, an error message will be printed. If the '-f' option is given, no error message will be given and file2 will be removed first.

Examples:

    mv test foo
    mv -f test foo
    mv test test,d2

The first example simply renames the file "test" as "foo". The second example deletes the file "foo" and then renames "test". The last example copies the file "test" from the current data drive to drive two and then deletes "test" from the current drive.

2.6.13 rm

rm file1 [file2] ...

Delete the specified file or files. If a file is locked, a message is displayed giving the name of the file which is locked.

Examples:

rm file1 file2
rm foo,s5

The first example deletes files "file1" and "file2" from the current data drive. The second example deletes the file "foo" from the disk in slot five. The drive number will be the same as the current data drive number.

2.6.14 run

run [arg1] [arg2] ...

Does a jsr to the starting address of the last file loaded after pushing a pointer to the argument vector and the number of arguments on the stack. Argv[0] will be the "run" string.

Example:

load tabset
A=0800 L=12F0
run 8

This example loads the program "settab" into memory. The SHELL displays the load address and length. The "run" command then calls hex 800 with the argument "8". The three lines are equivalent to typing:

tabset 8

all by itself.

2.6.15 save

save file [aN] [1N]

Saves a part of memory to a file on the specified device. If the starting address and length are not specified, the starting address and length of the last file "load"ed will be used. If N is begun with a '$', the value is interpreted as a hex value otherwise as decimal.

Examples:

save foo
save foo a$800 11000

The first example will save in a file called "foo", whatever the last program loaded or run. The second example will save a thousand bytes of memory starting at hex 800 in a file called "foo".

2.6.16 unlock

unlock file1 [file2] ...

Unlock the file on the specified slot, drive and volume. If any of slot, drive or volume are not given, they default to the current data values.

Examples:

unlock test1
unlock test1 test2 test3,d2

The first example unlocks file test1 on the current data disk. Example two unlocks files test1 and test2 on the current data disk and unlocks file test3 on drive two of the current data slot and volume.

2.7 Batch Facilities

Text files containing a series of SHELL command lines can be executed by simply typing the name of the text file followed by a carriage return. Note that the type of the file must be 'T'. All SHELL input is then taken from that file until the end is reached. SHELL command files may not be nested, but they may be chained. If a SHELL command line executes a second SHELL command file, the first command file is closed and forgotten. Lines beginning with the '#' character are ignored by the shell and can be used as comments.

When the SHELL is booted for the first time, the disk that the SHELL was booted from is searched for a file called ".PROFILE". If this file is found and is a text file, it will be executed immediately. This allow any special startup procedures to be automatically initiated.

SHELL command files may also be given up to 9 arguments. These arguments are referenced by the character '$' followed by the number of the argument to be used. Argument 0 is the name of the SHELL command file itself. For example, to link together several files, the following one line SHELL command file might be created:

    ln -o ln.out $1 $2 $3 $4 $5 $6 $7 $8 $9 sh65.lib

If the file was called "linkit", it could be used by typing:

    linkit f1.rel f2.rel f3.rel

If an argument does not exist, it is ignored.

There are two special "built-in" commands that the SHELL will only recognize when read from a SHELL command file. These commands are used for additional control over the processing of the commands in a SHELL command file.

2.7.1 loop

    loop

This command is used to start and end a loop in a SHELL command file. The command lines between the two loop statements will be executed once for each argument given to the SHELL command file. During the loop, two special arguments are available for use. '$#' will be replaced on any command line that it appears by the number of the current argument being processed. The two character sequence '$%' will be replaced by the current argument itself. The following is an example of a SHELL command file which will compile and assemble from one to nine files, one at a time.

```
        set -x -a
        loop
        # This is argument number $#, $%
        c65 -a -o $%.asm $%.c
        as65 -o $%.rel $%.asm
        loop
```

If the preceding lines were placed in a file called "compile", then the statement:

```
        compile test junk foo
```

would compile and assemble the three files "test.c", "junk.c", and "foo.c" into the corresponding ".rel" files and produce:

```
        loop
        This is argument 1, test
        c65 -a -o test.asm test.c
        as65 -o test.rel test.asm
        loop
        This is argument 2, junk
        c65 -a -o junk.asm junk.c
        as65 -o junk.rel junk.asm
        loop
        This is argument 3, foo
        c65 -a -o foo.asm foo.c
        as65 -o foo.rel foo.asm
        loop
```

2.7.2 set

```
        set [+-x] [+-a] [+-n]
```

Sets or clears one of three internal flags in the SHELL. Using '+' will clear the flag while '-' will set it. The flags are defined as follows:

x   echo command lines to the screen. Defaults to off.
a   abort the SHELL command file if a command or program exits with a non-zero value. Defaults to no abort.
n   parse the command lines, but do not execute them. Defaults to off.

Thus, to see each line being executed, the first line of a SHELL command file should be:

```
        set -x
```

To have a SHELL command file exit if an error occurs, include the line:

```
        set -a
```

The "set" command may only be executed within a SHELL command file.

2.8 Configuration

The basic Apple II is limited in it's ability to deal with upper and lower case and has a limited screen size. The SHELL contains device drivers which allow it to overcome these limitations to some degree. However, these same routines have been set up to take advantage of optional peripherals which greatly enhance the Apple's operation. There are two approaches to dealing with peripheral devices, writing custom routines to deal with one particular device or to write a general routine to handle a number of similar devices. The original versions of the SHELL device drivers were examples of writing custom routines. The current version contains general purpose routines for dealing with three devices, the keyboard, screen and printer.

The device routines make use of a table at a fixed location in the driver to handle the functional differences between different hardware configurations. This table can be modified by using the CONFIG program provided on the STARTUP diskette. A separate set of options is available for each device and are detailed in the following.

2.8.1 Keyboard

The first device is the keyboard. There are four variations of keyboard available. First, is a full upper and lower case keyboard as is available with the IIe or a keyboard enhancer. If this option is selected, no mapping is done at all. Second, is an Apple keyboard with the single wire shift key mod installed, while the third is an Apple keyboard without the SWSKM. Both of these options map characters from the keyboard to get the full range of ascii characters. Finally, it is possible to specify that the keyboard is a remote terminal. In this case, the driver will use the Pascal 1.0 entry point to the card that is assumed to be in slot 3. It will not do any mapping on the data received from the card. Also, since there is no status entry point, the ^S and ^C output control characters are not available. The ^C abort is still enabled during input.

2.8.2 Screen

The second device is the screen. There are three types of screen. First, the basic Apple screen with 40 columns and upper case only. Second, 40 columns with upper and lower case capability. Examples of this are the IIe and a II with a lower case adapter. Finally, there are the 80-column screens. All 80-column screens, remote and otherwise, are assumed to reside in slot 3 and are accessed by using the Pascal 1.0 output hook.

Not all 80-column screens are identical, and do not necessarily use the same control sequences to perform such functions as clearing the screen, moving the cursor and others. To minimize this problem, a table of control codes has been built into the SHELL device driver. This table is used by the ioctl() routine when performing the appropriate functions. See the

section on Device Ioctl for more information on its use.

The values in this table are already known for several devices by the CONFIG program. The devices whose values are known are the IIe, the Videx Videoterm, and the Smarterm. If a device is used which is not compatible with any of the above three cards, then the entries to the table must be provided by the user. The only programs which currently make use of the ioctl() screen calls are the screen editor VED, and the CONFIG program. For VED, the only required functions are cursor positioning, clear to end of line, and clear screen. CONFIG only uses the clear screen.

The following is a list of the entries in the table which will be followed by a discussion of each one. All values are in hex. If any function is not available the table entry should contain a zero.

TABLE ITEM	II	IIe	Videx	Smarterm	TVI-912
LEAD IN CHARACTER	00	00	00	00	1B
CURSOR MOTION	0F	1E	1E	1E	BD
CURSOR X/Y BYTE	20	20	20	20	A0
CLEAR SCREEN	1A	0C	0C	0C	1A
CLEAR TO END OF SCREEN	13	0B	0B	0B	D9
CLEAR TO END OF LINE	14	1D	1D	1D	D4
INSERT LINE AT CURSOR	00	00	00	00	C5
DELETE LINE AT CURSOR	00	00	00	00	D2
INSERT CHARACTER AT CUR	00	00	00	00	D1
DELETE CHARACTER AT CUR	00	00	00	00	D7
TURN INVERSE ON	00	0F	00	00	EA
TURN INVERSE OFF	00	0E	00	00	EB
SCROLL SCREEN UP	00	17	00	00	00
SCROLL SCREEN DOWN	0E	16	00	00	00
CURSOR UP	0B	00	1F	1F	0B
CURSOR RIGHT	0C	1C	1C	1C	0C
MOVE CURSOR HOME	1E	19	19	19	1E

LEAD IN CHARACTER

The lead in character is used by some terminals to signify that the next character is a control character. The example in the last column for the TVI-912 terminal uses a lead in character of $1B or the ESCAPE character. Whenever a lead in character is to be used in a control sequence, the high bit of the control character should be on in the table. This bit will be stripped out before the control character is sent to the device. For example, for the TVI, to clear the screen, the character to be sent is a ^Z or hex 1A with no lead in character. The entry for CLEAR SCREEN then is just 1A. To clear to the end of the screen, however requires the two character sequence ESC Y to be sent. Thus, the lead in byte is 1B (ESC), and the control byte is the hex value for Y (59) with the high bit turned on which is D9. If only a 59 were in the table, then only a 'Y' would be sent to clear to the end of the screen.

CURSOR MOTION and CURSOR X/Y BYTE

These two table entries are tied together. Normally to move the cursor, a control character is sent followed immediately by two bytes giving the X and Y coordinates. The entry for CURSOR MOTION is the control character. If a lead in is necessary, the high bit should be on.

The two bytes giving the X and Y coordinates are usually not zero based. Most terminals base these coordinates using a space as the base. Thus, coordinate zero is a space, coordinate one is a '!', coordinate two is a '"', etc. The CURSOR X/Y BYTE contains the value which is to be added to the X and Y coordinates before being sent to the device. The hex value for a space is 20, which is the entry in the table for most devices.

The high bit for this byte does not involve the lead in character. The high bit of the control byte was used for that. Instead, if the high bit of the X/Y byte is on, that indicates that the Y coordinate should be sent first, otherwise the X coordinate will be. Notice in the table that all the devices use the space with the X coordinate first, except for the TVI which uses the space with the Y coordinate first.

CLEAR SCREEN

This control clears the entire screen and places the cursor at the home position.

CLEAR TO END OF SCREEN

This control clears from the cursor to the end of the screen without moving the cursor.

CLEAR TO END OF LINE

This control clears from the cursor to the end of the line without moving the cursor.

INSERT LINE AT CURSOR

This control inserts a blank line at the cursor position, pushing all other lines down one. The cursor position is somewhere on the line.

DELETE LINE AT CURSOR

This control deletes the line the cursor is on and pulls all lines below up one, leaving a blank line at the bottom of the screen. The cursor position is somewhere on the line.

INSERT CHARACTER AT CURSOR

This control inserts a blank space at the cursor postion,

pushing the remaining characters of the line to the right one place. The last character is pushed off the screen and lost. The cursor remains stationary.

DELETE CHARACTER AT CURSOR

This control deletes the character under the cursor pulling the remaining characters of the line to the left one place. The last position is filled with a space. The cursor remains stationary.

TURN INVERSE ON

This control turns on inverse such that all following characters are displayed as black on white.

TURN INVERSE OFF

This control turns inverse off, such that all following characters are displayed as white on black.

SCROLL SCREEN UP

A blank line is inserted at the bottom of the screen and all other lines are moved up. The top line is lost. Cursor position before and after is not specified.

SCROLL SCREEN DOWN

A blank line is inserted at the top of the screen and all other lines are moved down. The bottom line is lost. The cursor position before and after is not specified.

This control is used by the VED editor when moving backwards through a file. If this control is not available, VED will repaint the screen half a screen back instead of scrolling. This function is supported in the 40 column mode and is supported by the IIe 80 column card as well. The Videx Videoterm firmware does not support this feature, but a small assembly language routine has been included as part of the screen driver which performs this function for the Videoterm only.

MOVE CURSOR UP

This control maintains the same horizontal position of the cursor, but moves it to the line above the one that it is on.

MOVE CURSOR RIGHT

Also called non-destructive right, this code moves the cursor right one position without changing any characters on the screen.

MOVE CURSOR TO HOME

This moves the cursor to the upper left corner of the screen without changing any characters on the screen.

While entering these values into the CONFIG program, backspace may be used if a mistake is made. Type return after each entry.

2.8.3 Printer

The last device is the printer. The printer is assumed to be driven by a peripheral card in slot 1 with firmware which supports the PR# basic protocol. The printer is initialized by placing the address of the card in the CSW vector in low memory and then calling the card. Normally the card then replaces the address in the CSW vector with the normal character output routine. The printer driver then sends a string of characters to initialize the firmware on the card. The default sequence is:

> ^I^Y^Y255N

which tells the card that the width is 255 and not to echo the characters to the Apple screen. It is not really necessary to change the control character to be something other than a ^I since tabs are expanded to spaces by the print driver. After the driver sends the initialization string, it saves the address in CSW for use when sending characters to the printer. When sending characters, the address is placed back in CSW and control passed to the firmware by jumping indirectly through CSW.

The printer has three other control modes. First, some printer card firmware requires that the high bit be on for characters sent to it. If so, the driver has a flag which will cause it to "or" in a hex 80 with each character before transmitting it to the firmware. Also, the print driver automatically converts newlines (LF) to carriage returns before transmitting to the firmware on the card. If the appropriate flag is set, the print driver will automatically send a line feed after each carriage return. Finally, when the printer is closed it is possible to have the print driver automatically send a form feed ($0C) character to the device. All these flags are set by answering the appropriate questions in the CONFIG program.

2.9 Memory Map

The following is a map of memory after the SHELL has been booted.

```
F800 - FFFF           - exact copy of same locations in ROM.
E000 - F7FF           - SHELL
D000 - DFFF  (2) - (bank 2) SHELL
D000 - DFFF  (1) - (bank 1) SHELL device drivers (see expansion)
C000 - D000           - Apple I/O space
AAA0 - BFFF           - DOS 3.3 file manager and RWTS routine
A800 - A9FF           - SHELL data and variable space
A0FE - A7FF           - DOS buffers (default 3)
???? - A0FD           - C program pseudo-stack
 800 - ????           - program code and data
 400 -  7FF           - Apple 40 column screen
 3FE -  3FF           - IRQ handler vector
 3FB -  3FD           - NMI handler
 3F8 -  3FC           - ctl-Y handler
 3F5 -  3F7           - APPLESOFT & handler
 3F4 -                - power-up byte
 3F2 -  3F3           - autostart reset handler
 3EF -  3F1           - autostart BRK handler
 3EE -                - unused
 3ED -                - SHELL reset counter
 3EA -  3EC           - unused
 3E3 -  3E9           - subroutine to locate RWTS parameter list
 3DC -  3E2           - subroutine to locate file manager parameter list
 3D9 -  3DB           - jump to RWTS
 3D6 -  3D8           - jump to DOS file manager
 3D3 -  3D5           - jump to SHELL cold start
 3D0 -  3D2           - jump to SHELL warm start
 3CD -  3CE           - jump from shell to float
 3CB -                - return value of program
 3C0 -  3CA           - reserved for expansion
 300 -  3BF           - free memory
 200 -  2FF           - SHELL command file argument storage
 100 -  1FF           - 6502 machine stack
  FE -   FF           - SHELL top of stack (HIMEM)
  FD -                - current maxfiles value
  FC -                - page number of locations containing pointer to DOS bu
  8F -   FB           - unused by SHELL
  80 -   8F           - register variables
  20 -   7F           - unused by SHELL other than normal monitor use
   8 -   1F           - general purpose pseudo-registers
   6 -    7           - interpreter program counter or secondary frame pointe
   4 -    5           - C frame pointer
   2 -    3           - C pseudo-stack
   0 -    1           - scratchpad locations
```

2.9.1 Bank 1 expansion:

```
D0A0 - D7FF      - program and data for devices
D095 - D09F      - reserved for expansion
D093 - D094      - address of keyboard map table
D091 - D092      - address of printer init string
D090 -           - length of printer init string
D08F -           - printer flags byte
D08E -           - special character to end input
D08D -           - special character to stop output
D08C -           - special character for CAPS LOCK
D08B -           - special character to exit program
D08A -           - screen length
D089 -           - screen width
D088 -           - tab width
D087 -           - screen flags byte
D086 -           - keyboard flags byte
D084 - D085      - address of the terminal characteristics array
D082 - D083      - address of output routine
D080 - D081      - address of input routine
D070 - D07F      - eighth device entry (UNUSED)
D060 - D06F      - seventh device entry (UNUSED)
D050 - D05F      - sixth device entry (UNUSED)
D040 - D04F      - fifth device entry (UNUSED)
D030 - D03F      - fourth device entry (UNUSED)
D020 - D02F      - third device entry (UNUSED)
D010 - D01F      - second device entry (PR:)
D000 - D00F      - first device entry (KB:)
```

2.10 Custom Device Drivers

Writing and interfacing custom device drivers for the SHELL is a task which should be undertaken only by someone familiar with assembly language and with the device being interfaced. The device driver must conform to the guidelines for device control or it will not operate correctly with the SHELL. More information regarding the structure, building and loading of custom drivers can be found in the Technical Information section of this manual.

Programs

SECTION 3 - PROGRAMS

.

3.1 C65 - Native Code Compiler

c65 [-abts] [-o file] [-Dtoken] [-Enn]
                              [-Xnn] [-Ynn] [-Znn] file.c

The Aztec C65 compiler is a true native code C compiler. C65 produces in-line assembly language code for all C statements with the following exceptions:

a) All floating point operations.
b) Multiplication, division, and modulus.
c) Shifts.
d) All pseudo-stack operations.
e) Switches.
f) Structure copies.

The code generated by the compiler uses a 16 bit pseudo-stack pointer kept in locations 2-3 of zero page. This stack is used for all local variable storage and for passing arguments to functions. The return address of function calls is also stored on the pseudo-stack. The 6502 machine stack is only used for temporary storage, thus fully recursive programming may be used without the limitations of the 6502 machine stack.

C65 makes use of the first thirty-two locations of zero page as work space and temporary registers. C65 also uses locations $80-$8F of zero page as user declared register variables. Up to eight "register" declarations are accepted within each function. Each routine which uses register variables automatically saves the locations it uses on the pseudo-stack and restores them when it exits. Chars, ints, unsigned ints and pointers may be declared as registers.

Use of register variables produces significantly smaller and faster code. The hidden overhead of saving and restoring register variables is minor compared to the gain in speed and code size.

The simplest use of the compiler is just:

c65 name.c

It is recommended that the file name end in ".c", but it is not necessary. C source statements found in the "name.c" file are translated to 6502 assembler source statements and written to a file named "$TMP.$$$". Then the compiler will automatically execute the AS65 assembler which will assemble the "$TMP.$$$" file and produce a relocatable object file called "name.rel". The "$TMP.$$$" file will then be deleted by the assembler.

The options available with C65 are listed below.

-a

If it is necessary to view the assembly language produced by C65, this option will force the compiler to leave the ouput in a file called "name.asm", where "name" is from the first part of the file being compiled. In this instance, the assembler will not be executed. For example:

c65 -a dbms.c

will leave the assembly language output in the file "dbms.asm".

-o

This option allows the user to specify the name of the output file. Normally, this can be used to specify the name of the relocatable object module as in:

c65 -o temp.rel dbms.c

which compiles "dbms.c" and then assembles the "$TMP.$$$" file and places the output of the assembler in "temp.rel".

When used with the "-a" option, it specifies the name of the assembly language file instead, as in:

c65 -a -o temp.asm dbms.c

which compiles "dbms.c" and place the assembly language in the file "temp.asm" and quits.

-b

Normally, when conditionals are evaluated, the compiler generates a test and a branch around a "jmp" instruction since it cannot know that the branch will be in range. For example:

```
        cmp     #45
        beq     .5
        jmp     .17
.5
```

This option will force the compiler to generate a direct branch instead of the branch and jump, as in:

```
        cmp     #45
        bne     .17
```

If the branch is too long, an error message will not be generated until the module is linked. Most of the library was compiled with this option.

-s

By default, AZTEC C expects that pointer references to members within a structure are limited to the structure associated with the pointer. However, to support existing source where this is not the case, the "-s" option is provided. If the "-s" is specified as a compile time option and a pointer reference is to a member name that is not defined in the structure associated with the pointer then all previously defined structures will be searched until the specified member is found. The search will begin with the structure most recently defined and search backward from there.

-t

The "-t" option will copy the C source statements as comments in the assembly language output file. Each C statement is followed by the assembly language code generated from that statement.

-D

This option allows a token to be entered into the macro definition table as being defined. This is most useful for controlling the conditional compilation of code. For example, if a section of code is to be included for a specific customer, it might be surrounded by an ifdef-endif combination:

```
#ifdef CUSTOM
     statement1;
     statement2;
     statement3;
#endif
```

When normally compiled, these statements would not be included, but when compiled with:

```
c65 -DCUSTOM prog.c
```

the statements would be compiled into the program. Multiple uses of the "-D" option are permitted on one command line.

There are four options for changing default internal table sizes.

-E

The "-E" option specifies the size of the expression work table. The default value for "-E" is 120 entries. Each entry uses 14 bytes. Each operand and operator in an expression requires one entry in the expression table. Each function and each comma within an argument list is an operator. There are some other rules for determining the number of entries that an expression will require. Since they are not straightforward and are subject

to change, they will not be defined here. The best advice is that
if a compile terminates because of an expression table overflow
(error 36), recompile with a larger value for "-E".

The following expression uses 15 entries in the expression
table:

a = b + function(a + 7, b, d) * x;

The following will reserve space for 300 entries in the
expression table:

c65 -E300 prog.c

There must be no space between the "-E" and the entry size.

-X

The "-X" option specifies the size of the macro (#define)
work table. The macro table size defaults to 2000 bytes. Each
"#define" uses four bytes plus the total number of bytes in the
two strings. The following macro uses 9 bytes of table space:

#define v   0x1f

The following will reserve 4000 bytes for the macro table:

c65 -X4000 prog.c

The macro table needs to be expanded if an error 59 (macro
table exhausted) is encountered.

-Y

The "-Y" option specifies the maximum number of outstanding
cases allowed in a switch statement. The default size for the
case table is 200 entries, with each entry using 4 bytes.

The following will use 4 (not 5) entries in the case table:

```
switch (a) {
case 0:
    a += 1;
    break;
case 1:
    switch(x) {
    case 'a':
        funct1(a);
        break;
    case 'b':
        funct2(b);
        break;
    }
    a = 5;
case 3:
```

```
        funct2(a);
        break;
    }
```

The following allows for 300 outstanding case statements:

c65 -Y300 prog.c

The size of the case table needs to be increased if an error 76 (case table exhausted) is encountered.

-Z

The "-Z" option specifies the size of the string literal table. The size of the string table defaults to 2000. Each string literal occupies a number of bytes equal to the number of characters in the string plus one (for the null terminator).

The following will reserve 3000 bytes for the string table:

c65 -Z3000 prog.c

The size of the string table needs to be increased if an error number 2 (string space exhausted) is encountered.

The name of the C source file must always be the last argument.

The AZTEC C native-code compiler is implemented according to the language description supplied by Brian W. Kernighan and Dennis M. Ritchie, in The C Programming Language. The user should refer to that document for a description and definition of the C language. This document has detailed areas where the AZTEC C compiler differs from the description in that book.

The reader who is not familiar with C and does not have a copy of the Kernighan and Ritchie book is strongly advised to acquire one. The book provides an excellent tutorial for learning and using C. The program examples given in the book, can be entered, compiled with AZTEC C and executed to reenforce the instruction given in the text.

The library routines defined in standard C that are supported by AZTEC C are identical in syntax to the standard. The library routines that are supported are defined in the library section of this manual. AZTEC C includes some extended library routines, that do not exist in the standard C, to allow access to native operating system functions. These are also described in the library section. The system dependent functions should be avoided in favor of the standard functions if there is or may be a requirement to run the software under different operating systems.

Information regarding interfacing with assembly language can be found in the Technical Information section of this manual.

3.2 CCI - Pseudo-code Compiler

        cci [-ats] [-o file] [-Dtoken] [-Enn]
                              [-Xnn] [-Ynn] [-Znn] file.c

        The Aztec CCI compiler is a pseudo-code compiler. CCI produces assembly language for a pseudo-machine that is interpreted by a 6502 assembly language program. The pseudo-machine makes use of the same pseudo-stack and registers as the native code produced by C65. Thus, there is no difficulty in mixing routines compiled using C65 with routines compiled with CCI.

        CCI differs from C65 in that it does not make use of the "register" type definition. The declaration is allowed, it is simply ignored.

        AZTEC CCI is invoked by the command:

        cci name.c

        It is recommended that the file name end in ".c", but it is not necessary. C source statements found in the "name.c" file are translated to pseudo-code assembler source statements and written to a file named "$TMP.$$$". Then the compiler will automatically execute the ASI assembler which will assemble the "$TMP.$$$" file and produce a relocatable object file called "name.int". The "$TMP.$$$" file will then be deleted by the assembler.

        The options available with CCI are the same as those for C65, with the exception of the "-b" option, which does not apply to the pseudo-code compiler. Please refer to the section on C65 for more information.

3.3   AS65 - 6502 Assembler

as65 [-c] [-l] [-ZAP] [-o file] file.a65

3.3.1   Overview

The AZTEC AS65 assembler is a relocating assembler which supports most of the standard MOS Technology mnemonics and is normally invoked by the command line:

as65 test.a65

The file "test.a65" is the assembly language source file. The filename does not have to end in ".a65". In this case, the relocatable object file produced by the assembler will be named "test.rel" where test is the same name as the prefix of the input filename. The type of the file in a CATALOG will be 'R'. There are several options to the assembler which are detailed below.

-o

An alternative object filename can be supplied by specifying the option "-o filename". The object file will be written to the filename following the "-o". The filename does not have to end in ".rel", it is, however, the recommended format.

-c

This option forces the assembler to make two passes through the source file. This allows most forward references to be resolved during the second pass. The overall result is that the object file size is significantly smaller since very few local labels need to be stored in the object module. This option was added primarily for the production of libraries, where size of the module is important. The overhead of reading the source file twice makes this option much less useful during normal compilation and assembly with one exception. If the "-b" option of C65 is used, using the "-c" option will detect a branch out of range without having to use the linker.

-l

This option generates a listing of the assembly language file. All opcodes are specified in the listing and all arguments that are known. Unknown arguments such as forward branches and addresses are represented as "XX". Using the "-c" and the "-l" together eliminates the "XX"'s in forward branches. The output is placed in a file with a ".lst" extension.

-ZAP

This option forces the assembler to delete the input file
after performing the translation. This option is used by C65 when
it automatically executes the assembler to delete the temporary
file "$TMP.$$$".

3.3.2 Syntax

The following defines the syntax for the AS65 assembler.

Statements

Source files for the AZTEC AS65 assembler consist of
statements of the form:

   [label]   [opcode] [argument]   [[;]comment]

The brackets "[...]" indicate an optional element.

Labels

A label consists of any number of alphanumerics starting in
column one. If a statement is not labeled, then column one must
be a blank or a tab or an asterisk. An asterisk denotes a comment
line. A label must start with an alphabetic. An alphabetic is
defined to be any letter or one of the special characters ' ' or
'.'. An alphanumeric is an alphabetic or a digit from 0 to 9. A
label followed by "#" is declared external. The AZTEC C compiler
places a '_' character at the end of all labels that it
generates.

Expressions

Expressions are evaluated from left to right with no
precedence as to operator or parentheses. Operators are:

            * -multiply
            / -divide
            + -add
            - -subtract
            # -constant
            = -constant
            < -low byte of expression
            > -high byte of expression

Constants

The default base for numeric constants is decimal. Other bases are specified by the following prefixes or suffixes:

BASE	PREFIX	SUFFIX
2	%	b,B
8	@	o,O,q,Q
10	null,&	null
16	$	h,H

A character constant is of the form 'character as in 'A.

Assembler Directives

The AZTEC AS65 assembler supports the following pseudo operations:

COMMON	block name	- sets the location to the selected common block.
CSEG		- select code segment.
DSEG		- select data segment.
END		- end of assembler source statements.
ENTRY	expr	- entry point of final module.
EQU	expr	- define label value.
FCB	expr	- define byte constant.
FCC	/expr/	- define byte string constant.
FDB	expr	- define double byte constant.
FUNC	label	- if label is not defined then it is declared external.
INSTXT	/file/	- the specified file is included at this point.
PUBLIC	label	- declares label to be external.
RMB	expr	- reserves expr bytes of memory with no particular value.
WEAK	expr	- define label value if not previously defined.

3.4 ASI - Pseudo-code Assembler

asi [-ZAP] [-o file] file.asm

The AZTEC ASI assembler is a relocating assembler and is invoked by the command line:

asi name.asm

The relocatable object file produced by the assembly will be named "name.int" where name is the same name as the name prefix on the ".asm" file. The type of the file in a CATALOG of the disk will be 'R'. An alternative object filename can be supplied by specifying "-o filename". The object file will be written to the filename following "-o". The filename does not have to end with ".int", it is, however, the recommended format. The file "name.asm" is the pseudo-code assembly language source file. The filename does not have to end in ".asm".

The "-ZAP" option forces the assembler to delete the input file after performing the translation. This option is used by CCI when it automatically executes the assembler to delete the temporary file, "$TMP.$$$".

The complete definition of the pseudo-code and the syntax are not currently available.

3.5 LN - Linker

            ln [-t] [-o outfil] [-r] [-b N] [-c N]
                                      [-d N] [-f infil] file.rel ...

        The AZTEC LN link editor will combine object files produced
by the AZTEC ASI pseudo-code assembler and/or by the AZTEC AS65
6502 assembler, select routines from object libraries created
with the MKLIB utility and produce an executable binary file.

        Supplied with the AZTEC C Compiler System are several object
libraries. In most cases one or more of these libraries must be
specified. To link a simple single module routine, the following
command will suffice:

            ln name.rel libname.lib

        The operand "name.rel" is the name of the object file. The
executable file created by LN will be named "name". The "-o"
option followed by a filename can be used to create an
alternative name for the LN output file.

        Several modules can be linked together as in the following
example:

            ln -o name mod1.rel mod2.int mod3.rel libname.lib

        Also several libraries can be searched as in the following:

            ln -o name mod1.rel mod2.int mine.lib libname.lib

        Libraries are searched sequentially in order of
specification. It is expected that all external references are
forward. One way to deal with the problem of routines that make
external reference to a routine already passed by the librarian
is the following:

            ln -o name mod1.rel mine.lib mine.lib libname.lib

        The link editor will read the "mine.lib" library twice. The
second time through it will resolve backward references
encountered on the first pass.

        Other options for the link editor include:

-t

        to create a symbol table for debugging purposes. The symbol
table file will have the same prefix name as the output file with
a suffix of ".sym".

-b address

        to specify a base address other than hex 800. The base

address is normally the lesser of the code start address and the data start address, but may be lower than either. The "base address" is assumed to be in hex.

-c address

to specify a starting address for the code portion of the output. The default is the base address + 3. The first three bytes are usually occupied by a jump instruction to system initialization code. It is assumed that the code starting address is specified as a hex number.

-d address

to specify a data address. Data is usually placed immediately at the end of the code segment.

The three preceding options are usually used when producing ROMable code or for similar reasons. More information on ROMable code can be found in the Technical Information section of this manual. These options were used to link the SHELL so that the data was located outside the language card. This allowed the language card to be write protected. The command to link the SHELL was basically:

ln -o SHELL -b A7FD -d A800 -c D000 -f shell.lnk

The base address was set at three below the data since the linker automatically places a "jmp" to the start of the code at the base address unless the base and the code address are the same.

-f filename

to merge the contents of "filename" with command line arguments. More than one specification of "-f" can be supplied. There are several advantageous uses for this command. The most obvious is to supply the names of modules that are commonly linked together. All records in the file are read. There is no need to squeeze everything into one record.

-r

This option is used to inform the linker that the modules being linked are the root segment of a program with overlays. With this option, a file with a ".rsm" extension will be produced which is used in linking the overlays. More information on overlays can be found in the Technical Information section of this manual.

3.6 MKLIB

           mklib [-atxr] [-o library] [.] module1 module2 ....

        This program creates libraries which can be used by the
linker, LN, in a very efficient manner. Each module is
individually rearranged to make the linking process as fast as
possible. In addition, a dictionary of global variables which are
in the modules is automatically created as part of the library.
This dictionary is used by the linker so that it only looks at
those modules that it needs to.

        Note that a library may be specified as a module. In that
case, all the modules in the library are copied into the new
library. This can be used to combine several libraries into a
single library.

        There are several options to MKLIB which are detailed below.
The simplest use, however is to create a new library, as in:

           mklib mod1.rel mod2.rel mod3.rel

which creates a library called "libc.lib" with the three named
modules. As a convenience, if the number of modules to be added
to a library are large, the "." option can be used. When the
program processes the "." in the argument list, it switches its
argument parsing to the standard input. Thus, if a file
containing the names of all the modules to be linked has been
created called "infil", then

           mklib . < infil

will create a library called "libc.lib" with the modules named in
"infil".

-o library

        This option specifies the name of the library to be created
or to be modified. The default name of "libc.lib" is used if this
option is not specified. For example:

           mklib -o mylib.lib file1.rel file2.int . < infil

places the output in "mylib.lib".

-t

      This option lists the names of the modules in the library.
Note that a module may contain several functions and that the
name of the module may have nothing to do with the names of the
functions within that module. Module names are derived from the
names of the files used to create the library. As an example:

        mklib -t -o mylib.lib

will list the module names of "mylib.lib".

-r

      This option copies the library module by module. If a module
name matches the name of one of the modules specified as an
argument, the module is not copied from the library, but from the
object file instead. This process continues until the end of the
library is reached. At that point, the remaining module names are
appended to the library. The original library is deleted, and the
new library renamed.

-a

      This option appends the named modules to the end of the
library. All modules specified will be appended. However, in
order to update the dictionary properly, the library will be
copied in the process.

-x

      This option extracts the named library modules from the
library into individual files. If no module names are specified,
all modules are extracted.

3.7 VED - Screen Editor

ved [-tn] [file]

VED is a screen oriented text editor written in C for use with the Aztec C65 system. The source to VED is included in the archive "VEDSRC.ARC". VED is not a particularly fast or smart editor, but it does get the job done. If VED is invoked with a file name, that file will be loaded into the memory buffer, otherwise it will be empty. VED does all its editing in memory and is thus limited in the size of files that it will edit. In VED, the memory buffer is never completely empty. There will always be at least one newline in the buffer.

The "-t" option specifies that a different tab size should be used. Normally VED will use the current system value, but this may be overridden with this option, as in:

ved -t8 file.a65

which is useful since C programs work well with a tab size of four, but assembly language works better with a tab size of eight.

VED has a 1000 character limit on the size of a line. If a line is longer than the width of the screen, it will wrap to the next line. If a line starts at the bottom of the screen, and is too wide to fit, the line will not be displayed. Instead, the '@' character will be displayed. Likewise, at the end of the file, all lines beyond the end will consist only of a single '-' on each line.

A number of commands take a numeric prefix. This prefix is echoed on the status line as it is typed.

The normal mode of VED is command mode. During command mode, there are a number of ways to move the cursor around the screen and around the whole file.

```
newline    - move to the beginning of the next line.
-          - move to the start of the previous line.
space      - move to the next character of the line.
backspace  - move to the previous character.
0          - move to the first character of this line.
$          - move to the last character of this line.
h          - move to the top line of the screen.
l          - move to the bottom line of the screen.
b          - move to the first line of the file.
g          - move to the n'th line of the file.
/string    - move to the next occurrence of 'string'.
```

When the cursor is in the appropriate spot, there are two commands used to delete existing text.

x   - delete the n character under the cursor up to but
               not including the newline.
dd   - delete n lines starting with the current line.

Note that deleting the last character on the line (newline character) causes the following line to be appended to the current line.

To add new text, hitting the 'i' key will cause the top line of the screen to indicate that you are now in <INSERT> mode. To exit insert mode, type ESCAPE (unless the CAPS LOCK mode is enabled, in which case type control Q). To insert a control character which means something special to VED into a text file, first type control-v followed by the control character itself. Control characters are displayed as '^X', where X is the appropriate character.

Typing 'o' will cause a new line to be created below the current line, and the cursor will be placed on that line and the editor placed into <INSERT> mode.

There are three commands used for moving text around. These commands make use of a 1000 character yank buffer. The contents of this buffer is retained across files.

yy   - yank n lines starting with the current line
               into the yank buffer.
yd   - yank n lines starting with the current line
               and then delete them.
p   - "put" the lines in the yank buffer after the
               current line. The yank buffer is not modified.

The 'z' command redraws the screen with the current line in the center of the screen. The 'r' command replaces the character under the cursor with the next character typed.

When in command mode, if the ':' key is hit, a ':' will be displayed on the status line. At this point, a number of special file-related commands may be given.

:f   - displays info about the current file.
:w file   - writes the buffer to the specified file name.
:w   - writes the buffer to the last specified file.
:e file   - clears the buffer and reads the named file.
:e! file   - clears the buffer and reads the named file
               even if the file was modified.
:r file   - reads the named file into the buffer.
:q   - exits the editor.
:q!   - exits editor even if the file was modified.

As can be seen VED protects from accidentally destroying the work being edited by preventing exiting or editing another file if the current file has been modified. If the file has been written using the ":w" command, the modified flag will be cleared.

VED will only edit Apple text files. Binary files will not be edited.

3.8   ARCH - Source Archive Utility

arch -[clvxa]o archive [-f infil] [file1] [file2] ...

This program is used to create and manipulate archive files. Archive files are used as a convenient means of collecting source modules together. The 'o' option must always be used to specify the name of the archive itself. Only one of the options 'lxa' may be specified. The 'v' option is a modifier for the 'xa' options and causes them to print each file name they act upon. The remaining options are detailed below.

-l

This option lists the named files in the archive, giving the name and size of each. If no file names are specified, all of the files in the archive are listed. For example:

arch -lo progsrc.arc

will list the names and sizes of the files in the "progsrc" archive.

-a

This option specifies that the named files be appended to the end of the archive. If the 'c' option is given as well, the archive is truncated before adding the files.

-x

This option extracts the named files from the archive. If no file names are given, all the files in the archive are extracted from the archive. The archive is not modified.

-f file

This option forces the ARCH program to read the named "file" for the names of the files to be placed in the archive. All the lines in the file are read, and more than one file name may be placed on each line.

Different types of files may be freely intermixed within an archive file.

3.9 OD - Hex Dump Utility

od [+nnn[.]] file1 [file2] [file3] ...

This program performs a binary dump in hex and ascii of the specified file to the standard output. The program continues until the end of the file and then dumps the next file if any. If the optional argument "+nnn" is supplied, "nnn" is used as an offset into the file where the dumping is to start. If "nnn" is followed by a '.', it is treated as a decimal number, otherwise it is considered to be a hex value. Each file will be dumped starting from the last offset argument encountered. Thus, an offset of "+0" will cause the files which follow it to be dumped from their beginning.

For example:

> od +16b oldtest newtest +0 junk
> od +1000. tstfil

3.10 CMP - Byte for Byte File Compare

cmp [-l] file1 file2

     This program compares two files on a character by character basis. When it finds a difference, a message is printed giving the offset from the beginning of the file. The program will normally stop after the first difference, unless the "-l" option is given. If the "-l" option is specified, CMP will list all differences in the format:

decimal offset   hex offset   file1 value   file2 value

If no difference is found, the program will exit without saying anything.

    For example:

     cmp -l otst ntst

        10   a: 00 45
        100   64: 1a 23

and

     cmp otst ntst

     Files differ character 10.

3.11 NM - Name List Generator

         nm [-sunago] file1.int [file2.rel] ...

     This utility operates only on the relocatable object files
which are the output of the two assemblers, AS65 and ASI. This
program prints the symbol table (name list) of each object file.
The output consists of a symbol name preceded by the value of
that symbol. Between the symbol name and its value is a character
indicating the type of symbol. The characters used are:

     A - absolute
     T - program text
     D - initialized data
     C - common
     R - reference to common
     E - expression
     U - undefined
     W - weak definition

     The options available are:

     -s   Display only the size of the code and data.
     -g   Print only global (external) symbols.
     -u   Print only undefined symbols.
     -n   Sort numerically.
     -a   Sort alphabetically.
     -o   When multiple file names are given, each name is
             printed before the name list for that file. When
             this option is given, the file name is printed
             at the beginning of each line.

     For example, to see the size of several modules:

         nm -s mod1.rel mod2.int mod3.rel

or to see the undefined global symbols sorted in alphabetic
order:

         nm -uga mod1.rel mod2.int mod3.rel

3.12 TABSET

tabset [newsize]

This program displays the current setting of the tab width
parameter of the SHELL. If the argument is specified, the tab
width parameter is set to that value. In that case, both the old
and the new value are displayed. The parameter is stored in
location $D088 of bank 1 of the ram card.

3.13 CONFIG

config

This program takes no parameters as it is completely
interrogative. CONFIG is used to alter the SHELL's device driver
tables and thus make use of any non-standard peripherals. More
information on the use of the CONFIG program can be found in the
SHELL section of this manual.

3.14 LDEV

ldev file

This program replaces a former "built-in" SHELL command.
LDEV loads the named file into bank 1 of the RAM card and is used
when loading a new or custom set of device drivers. The format of
the device driver module is the same as that previously used.
However, to take advantage of new features such as the settable
tab width parameter, custom drivers will need to be incorporated
into the new SHELL drivers. More information is available in the
SHELL section and in the Technical Information section of this
manual.

Libraries

SECTION 4 - LIBRARIES

4.1 Introduction

The libraries provided with the Aztec C65 system can be divided into four logical groupings. These groups are the standard I/O, system I/O, utility, and math/floating point libraries. The source to all the libraries with the exception of the math library are provided with the system as archives. The compiled object modules are supplied in three libraries. However, six library files are supplied with the system, three compiled with C65 and three compiled with CCI.

The first library is the floating point library. This library contains the floating point emulation routines and all of the transcendental math functions. This library must be specified if any floating point operations are performed in any module being linked. If the library is not specified, the linker will abort with the symbol ".fltused" undefined. This library must be specified before the regular library for successful operation. The names of this library are FLT65.LIB and FLTINT.LIB which correspond to the C65 compiled version and the CCI compiled version.

The remaining two libraries are similar in function. The primary difference between the libraries involves their use of the SHELL. Since the SHELL contains many of the system I/O routines and a number of the utility routines, including the pseudo-code interpreter, these routines are not included in one of the libraries. Instead, a set of dummy addresses is included which provide a link to the routines within the SHELL. Programs which use the SHELL vector are smaller and therefore take less disk space and load faster. These programs also make use of the configured SHELL device drivers. The names of the SHELL libraries are SH65.LIB and SHINT.LIB.

The non-SHELL library contains all the routines in the SHELL library as well as all the routines used by the SHELL. This library is known as the stand-alone library, since programs linked with this library can be run without the SHELL in a normal DOS environment. The one significant difference, other than size, of programs linked with this library is that of console I/O. The I/O drivers supplied with the stand-alone library are not the same as those contained in the SHELL. The calling format and use is the same, but the actual routines are much simpler. More information on this subject can be found in the Technical Information section of this manual. The names of the stand-alone libraries are SA65.LIB and SAINT.LIB.

The differences between the libraries compiled with C65 and CCI are minimal, mostly relating to size and speed. Any program may be linked with either library without any hesitation or special procedures.

4.2 Summary

4.2.1 Standard I/O

agetc	(stream)	ASCII version of getc
aputc	(c, stream)	ASCII version of putc
clearerr	(stream)	clears the error flag on stream
exit	(return)	flushes and closes all streams
fclose	(stream)	closes an I/O stream
feof	(stream)	check for eof on stream
ferror	(stream)	check for error on stream
fflush	(stream)	write out buffered data to stream
fgetc	(stream)	gets a single character from stream
fgets	(buffer, max, stream)	reads line from stream to buffer
fileno	(stream)	returns the fd associated with strea
fopen	(name, how)	opens file name according to how
fprintf	(stream, format, argl, ...)	writes formatted print to stream
fputc	(c, stream)	writes character c to stream
fputs	(cp, stream)	writes string cp to stream
fread	(buf, sz, cnt, stream)	reads cnt items from stream to buf
freopen	(name, mode, stream)	switches stream to new file
fscanf	(stream, cntrl, pl, ...)	converts input string from stream
fseek	(stream, pos, mode)	positions stream to pos
ftell	(stream)	returns current file position
fwrite	(buf, sz, cnt, stream)	writes cnt items from buf to stream
getc	(stream)	gets a single character from stream
getchar	()	gets a single character from std
gets	(buffer)	reads a line from stdin
getw	(stream)	gets a word from stream
printf	(format, argl, ...)	writes formatted data to stdout
putc	(c, stream)	writes character c to stream
putchar	(c)	writes character c to stdout
puts	(cp)	writes string cp to stdout
putw	(w, stream)	writes a word w to stream
rewind	(stream)	position stream at beginning
setbuf	(stream, buf)	force stream to use buf
scanf	(cntrl, pl, ...)	converts input string from stdin
sprintf	(cp, format, argl, ...)	formats data into string cp
sscanf	(cp, cntrl, pl, ...)	converts input string cp
ungetc	(c, stream)	pushes c back into stream

4.2.2 System I/O

_exit	(return)	returns contol to operating system
catalog	(slot, drive, volume)	do a "CATALOG" of the disk
chmod	(name, how)	lock or unlock file name
close	(fd)	closes file fd
creat	(name, mode)	creates a file of type mode
ioctl	(fd, cmd, arg)	perform special I/O function
lseek	(fd, pos, mode)	positions file fd according to mode
open	(name, rwmode)	opens file according to rwmode
read	(fd, buf, size)	reads size bytes from file fd to buf
rename	(oldname, newname)	renames a disk file
unlink	(filename)	deletes a disk file
write	(fd, buf, size)	writes size bytes from buf to file f

4.2.3 Utility Routines

```
alloc     (size)                       allocates size bytes
atof      (cp)                         converts ASCII to floating
atoi      (cp)                         converts ASCII to integer
atol      (cp)                         converts ASCII to long
blockmv   (dest, src, size)            moves size bytes from src to dest
calloc    (nelem, elsize)              allocates space for nelem*elsize
clear     (area, size, value)          initialize area to value
execl     (prog, arg1, arg2, ...)    executes prog with args
format    (func, format, argptr)       outputs formatted data using func()
free      (addr)                       frees the space at addr
ftoa      (m, cp, prec, type)          converts floating to ASCII
htoi      (cp)                         converts ASCII hex to integer
index     (cp, c)                      finds c in string cp
isdigit   (c)                          checks for digit 0...9
islower   (c)                          checks for lower case a...z
isspace   (c)                          checks for white space
isupper   (c)                          checks for upper case A...Z
malloc    (size)                       allocates size bytes
rindex    (cp, c)                      finds c in string cp backwards
rwts      (tr,se,buf,cmd,sl,dr,vol)    read or write a sector from disk
settop    (size)                       bumps top of program memory by size
stcrat    (str1, str2)                 appends string 2 to the end of stri:
strcmp    (str1, str2)                 compares string 1 with string 2
strcpy    (str1, str2)                 copies string 2 to string 1
strlen    (str)                        returns length of string
strncat   (str1, str2, n)              appends at most n character
strncmp   (str1, str2, n)              compares at most n characters
strncpy   (str1, str2, n)              copies at most n characters
system    (str)                        SHELL executes string str
tolower   (c)                          converts to lower case
toupper   (c)                          converts to upper case
```

4.2.3 Math Routines

```
acos      (x)                          inverse cosine of x (arcos x)
asin      (x)                          inverse sine of x (arcsin x)
atan      (x)                          inverse tangent of x (arctan x)
atan2     (x, y)                       arctangent of x divided by y
cos       (x)                          cosine of x
cosh      (x)                          hyperbolic cosine of x
cotan     (x)                          cotangent of x
exp       (x)                          exponential function of x
log       (x)                          natural log of x
log10     (x)                          logarithm base 10 of x
pow       (x, y)                       raise x to the y'th power
random    ()                           random number from 0 to 1
sin       (x)                          sine of x
sinh      (x)                          hyperbolic sine of x
sqrt      (x)                          square root of x
tan       (x)                          tangent of x
tanh      (x)                          hyperbolic tangent of x
```

4.3 Standard I/O

The standard I/O library is based on the set of routines developed for use on UNIX operating systems. Use of standard I/O guarantees compatibility at the source leve with both UNIX and other Aztec C systems. Standard I/O is actually a set of routines which use the system I/O routines to perform the actual input and output. Standard I/O is fairly efficient when doing character at a time file I/O, since the data is buffered and there is no need to call DOS for each byte.

The three file descriptors 0, 1, and 2, which are pre-opended, are represented in the STDIO library by stdin, stdout, and stderr. These are pre-opened streams which are assigned to file descriptors 0, 1, and 2.

The buffer size used defaults to 256. The buffer space is not allocated until the first attempt to get or put a byte. Thus, the size of the buffer can be changed between the call to fopen and the first use of the stream as follows:

        FILE *fp;

        fp = fopen("textfile", "r");
        fp->bufsiz = 2048;

The buffer is allocated using the malloc() routine, and released when the file is closed by using the free() routine.

To use the standard I/O package, you should insert the statement:

                #include   "stdio.h"

into your programs to define the FILE data type and miscellaneous other things needed to use the functions.

There is one significant difference between the implementation of standard I/O on UNIX and in the ACCS. In the UNIX operating system, the end of line delimiter is the newline character which is normally specified as a backslash followed by the letter n, as in '\n'. This corresponds to a hex A, or line feed character. The apple, on the other hand, normally uses a hex D, or carriage return to end a line. This presents a problem.

The solution that has been adopted is to provide "raw" getc and putc routines as well as ASCII versions called agetc and aputc. The ASCII versions translate newlines to carriage returns on output, and carriage returns to newlines on input. The following routines do no translation:

getc	putc	fgetc	fputc
fread	fwrite	getw	putw
ungetc			

The following routines do the translation:

agetc	aputc	
gets	fgets	
printf	fprintf	sprintf
scanf	fscanf	sscanf
puts	fputs	
getchar	putchar	

In the following section, each routine will be described by specifying the routine name, the arguments and their types, the language in which the routine is written, the action performed by the routine and the return values if any.

agetc   **6502 Assembly**

```
agetc(stream)
FILE *stream;
```

This is an ASCII version of getc() which recognizes either carriage return ('\r') or line feed ('\n') or a carriage return/line feed sequence and maps it to newline ('\n'). End of file is returned as EOF. This routine provides a uniform way of reading ASCII data across several different systems.

aputc   **6502 Assembly**

An ASCII version of putc() which operates in the same manner as putc(). However, when a newline ('\n') is put into the file, an end of line sequence is written to the file (carriage return on Apple DOS).

clearerr   **C**

```
clearerr(stream)
FILE *stream;
```

Clearerr resets the error indication on the named stream.

exit   C

    exit(n)
    int n;

    Flushes and closes all open steams. Returns to the
    operating system with n as the return code. N is used
    to specify an error return to the SHELL.

fclose   C

    fclose(stream)
    FILE *stream;

    Flushes any data not yet written out and closes the
    named stream. EOF is returned if any errors occur,
    otherwise 0.

feof   C Macro

    feof(stream)
    FILE *stream;

    Returns non-zero if end of file has been encountered on
    the named stream.

ferror   C Macro

    ferror(stream)
    FILE *stream;

    Returns non-zero if an error has occurred reading or
    writing on the named stream. The error indication will
    remain with the stream until cleared using clearerr or
    until the stream is closed.

fflush   C Macro

    fflush(stream)
    FILE *stream

    If the steam has been opened for write, any data which
    has been buffered, is written out. EOF is returned if
    any errors occur, otherwise 0. (The real work is done
    by _flsh.)

fgetc   6502 Assembly

    fgetc(stream)
    FILE *stream;

    Returns the next character from the named stream. EOF
    is returned on end of file and is distinguishable from
    the regular data.

fgets
<div align="right">C</div>

```
char *fgets(buf, limit, stream)
char *buf;
int limit;
FILE *stream;
```

Reads n-1 characters or up to a newline character from the named stream. The last character read is followed by a null character. Fgets returns its first argument.

fileno
<div align="right">C Macro</div>

```
fileno(stream)
FILE *stream;
```

Returns the file descriptor used by the system I/O routines associated with the named stream. Note that intermixing standard I/O calls and system I/O calls on the same file may not have the desired effect because of the buffering done by standard I/O.

fopen
<div align="right">C</div>

```
FILE *fopen(name, mode)
char *name, *mode;
```

Opens a file or device according to "mode" and returns a pointer to an open I/O stream which can be used with other standard I/O functions. Name is a pointer to a string containing the name of the device or file in the same form as used by the SHELL.

NULL is returned if there is an error opening the file. The valid values for "mode" are:

"r"   The file is opened for reading only.
"r+"   The file is opened for reading and writing.
"rx"   The file is opened for reading, both the data
          and execution drives are searched.

"w"   The file is opened for writing. If "name" exists,
          it is deleted and a new one is created. The
          file created is a text file.
"wb"   Same as "w" except a binary file is created.
"wr"   Same as "w" except an object file is created.
"w+"   Same as "w" except the file is opened for reading
          and writing.
"wb+"   Same as "w+" except a binary file is created.
"wr+"   Same as "w+" except an object file is created.

fprintf   **C**

```
fprintf(stream, format, arg1, arg2, ...)
FILE *stream;
char *format;
```

Formats data according to format and writes the result
to the named stream. Formatting is done as described in
Chapter 7, Input and Output, of The C Programming
Language.

fputc   **6502 Assembly**

```
fputc(c, stream)
int c;
FILE *stream;
```

Appends the character c to the named stream. It returns
EOF on error, otherwise it returns the character
written.

fputs   **C**

```
fputs(str, stream)
char *str;
FILE *stream;
```

Writes the null-terminated string pointed at by str to
the named stream. EOF is returned on error.

fread   **C**

```
fread(buf, size, nitems, stream)
char *buf;
unsigned size;
int nitems;
FILE *stream;
```

Reads 'nitems' of length 'size' at the address pointed
to by 'buf' from the named stream. Returns the number
of completely read items.

freopen   **C**

```
FILE *freopen(name, mode, stream)
char *name, *mode;
FILE *stream;
```

Substitutes the named file in place of the open stream
after closing it. It returns the stream. This is
normally used to attach the pre-opened constant names,
stdin, stdout, and stderr to specified files.

fscanf   **C**

        fscanf(stream, control, ptr1, ptr2, ...)
        FILE *stream;
        char *control;
        int *ptr1, *ptr2, ...;

        Formats data according to control. Data is read from
        stream file. Formatting is done as described in chapter
        7, Input and Output, of The C Programming Language.

fseek   **C**

        fseek(stream, offset, how)
        FILE *stream;
        long offset;
        int how;

        Positions the named stream according to how and offset.
        If how is 0, file is positioned offset bytes from the
        beginning of the file. If how is 1, file is positioned
        offset bytes relative to the current position in the
        named stream. Fseek returns EOF if an error occurs.

ftell   **C**

        long ftell(stream)
        FILE *stream;

        Returns the current position in the named stream.

fwrite   **C**

        fwrite(buf,size,nitems,stream)
        char *buf;
        unsigned size;
        int nitems;
        FILE *stream;

        Writes 'nitems' of length 'size' from the address
        pointed to by 'buf' onto the named stream. Returns the
        number of completely written items.

getc   **6502 Assembly**

        getc(stream)
        FILE *stream;

        Returns the next character from the named stream. EOF
        is returned on end of file and is distinguishable from
        the regular data.

getchar   **C, C Macro**

>   getchar()
>
>   Returns the next character from the standard input, as
>   in agetc(stdin). This routine is defined as both a
>   macro and a routine, for the common case where
>   "stdio.h" is not included.

gets   **C**

>   char *gets(buf)
>   char *buf;
>
>   Reads characters from the standard input until end of
>   file or a newline character are received. The newline,
>   if received, is not placed in the buffer. The last
>   character received is followed by a null character.
>   Gets returns its argument unless an end of file is
>   received before any other characters are received, in
>   which case it returns 0.
>
>   If the standard input is connected to a character
>   device such as a keyboard, the characters, backspace
>   and control-X, are interpreted as character and line
>   delete respectively.

getw   **C**

>   getw(stream)
>   FILE *stream;
>
>   Returns the next word (two bytes) from the named
>   stream. The constant EOF is returned on end of file or
>   an error. However, since EOF is a valid integer, feof
>   and ferror must be used to determine the success of the
>   call.

printf   **C**

>   printf(format, arg1, arg2, ...)
>   char *format;
>
>   Formats data according to format and writes the result
>   to the standard output stream, stdout. Formatting is
>   done as described in Chapter 7, Input and Output, of
>   The C Programming Language.

putc   6502 Assembly

        putc(c, stream)
        char c;
        FILE *stream;

        Appends the character c to the named stream. It returns
        EOF on error, otherwise it returns the character
        written.

putchar   C, C Macro

        putchar(c)
        char c;

        Writes the character c to the standard output, as in
        putc(c, stdout). This routine is defined as both a
        macro and a routine, for the common case where
        "stdio.h" is not included.

puts   C

        puts(str)
        char *str;

        Writes the null-terminated string pointed at by str to
        the standard output. Unlike fputs, puts appends a
        newline to the string. EOF is returned on any errors.

putw   C

        putw(w, stream)
        int w;
        FILE *stream;

        Writes the word w to the named stream. Putw returns the
        word written. If an error occurs, it returns EOF which
        is a good integer, so ferror shoud be used to detect
        errors.

rewind   C

        rewind(stream)
        FILE *stream;

        Positions the named stream at its beginning, as in
        fseek(stream, OL, 0).

setbuf   C

```
setbuf(stream, buf)
FILE *stream;
char *buf;
```

Setbuf is used to specify that the buffer buf be used
instead of a system allocated one. Setbuf is used on an
open stream before it has been read or written. If buf
is the constant NULL, I/O will be unbuffered.

scanf   C

```
scanf(control, ptr1, ptr2, ...)
char *control;
int *ptr1, *ptr2, ...
```

Formats data according to control. Data is read from
the standard input. Formatting is done as described in
Chapter 7, Input and Output, of the The C Programming
Language.

sprintf   C

```
sprintf(str, format, arg1, arg2, ...)
char *str, *format;
```

Formats data according to format and stores the null-
terminated result in the buffer pointed at by str.
Formatting is done as described in Chapter 7, Input and
Output, of The C Programming Language.

sscanf   C

```
sscanf(str, control, ptr1, ptr2, ...)
char *str, *control;
int *ptr1, *ptr2, ...
```

Formats data according to control. Data is taken from
"str". Formatting is done as described in chapter 7,
Input and Output, of The C Programming Language.

ungetc   C

```
ungetc(c, stream)
char c;
FILE *stream;
```

Pushes the character c back on the named stream. The
character will be returned by the next getc call on
that stream. Normally returns c and returns EOF if c
cannot be pushed back. Only one character of pushback
is guaranteed and EOF cannot be pushed back.

4.3 System I/O

The system I/O routines are really just an interface to the DOS file manager and to the individual device drivers. A file or device is accessed by first using the "open" library routine. This routine returns an integer from 0 through 9. This value is used in subsequent "read", "write" and "close" calls and is called a file descriptor.

When a program is initiated from the SHELL, the file descriptors 0, 1 and 2 are all pre-opened and default to the keyboard and screen. This is a carry-over from the UNIX operating system convention of using file descriptor 0 for standard input, file descriptor 1 for standard output, and file descriptor 2 for the standard error output. These file descriptors may be closed and reopened to other devices or files. If the program is called using the '<' and/or '>' I/O redirection from the SHELL. The SHELL opens file descriptor 0 to the file name after the '<' and opens file descriptor 1 to the file name after the '>'.

The following is a description of each of the system I/O library calls.

_exit   C

        _exit(n)
        int n;

        Returns control to the operating system command parser.
        after closing all open files. Any non-zero value 'n'
        is considered to be an abort by the SHELL command file
        with the abort flag turned on.

catalog   C

        catalog(sdlot, drive, volume)
        int slot, ddrive, volume;

        The specified slot, drive and volume are cataloged.
        Return is zero, or the negative error number if an
        error occurs.

chmod   C

        chmod(name, how)
        char *name;
        int how;

        If how is 0, the file specified by name is unlocked.
        If how is 1, the file is locked. Return is zero, or
        the negative error number if an error occurs.

close   C

```
close(fd)
int fd;
```

The file or device specified by fd is closed.

creat   C

```
creat(name, mode)
char *name;
int mode;
```

Creat is used to open a file for writing which may or may not exist. If the file does not exist, the file is created with type 'mode', where 0 is an Apple text file, 4 is a binary file, and 16 is an object file. If the file does exist, the file is deleted. In either case, the file is opened for write only and the file descriptor returned. If the creat fails, the value returned is negative. This value indicates the type of error which occurred.

ioctl   C

```
ioctl(fd,   cmd, arg)
int fd, cmd, arg;
```

This function has different effects depending upon which device is specified by fd. Command numbers less than 100 are reserved for present and future definition by Manx Software Systems. Current definitions for supported devices may be found in the Device Ioctl section of this manual.

lseek   C

```
long lseek(fd, pos, how)
int fd;
long pos;
int how;
```

Positions the file specified by fd according to how and pos. If how is 0, the file is positioned at pos bytes from the beginning. If how is 1, the file is positioned pos bytes from the current position, where pos may be negative. In any case, the new position is returned. If an error occurs, a negative error number is returned.

open   C

```
open(name, mode)
char *name;
int mode;
```

Open is used to get access to the file or device 'name' and returns a file descriptor upon a successful open. The value of mode is 0 if the file is to be opened for read only, 1 if opened for write only, and 2 if opened for reading and writing. When a file is opened, it is positioned at the beginning.

Under the SHELL, an open mode of -1, will open for reading the same as mode 0, but will look on the current execution drive, and not the current data drive.

If the open fails, the value returned is negative and indicates the type of error which occurred.

read   C

```
read(fd, buf, cnt)
int fd;
char *buf;
int cnt;
```

This routine attempts to read 'cnt' characters into the buffer whose address is given by 'buf' from the file or device specified by 'fd'. The actual number of characters read is returned. A zero is returned if at the end of a file and a negative value if an error occurs.

rename   C

```
rename(old, new)
char *old, *new;
```

The file named 'old' is renamed as 'new'. Return is zero, or the negative error number if an error occurs.

unlink   C

```
unlink(name)
char *name;
```

The named file is deleted. Return is zero, or the negative error number if an error occurs.

write   C

```
write(fd, buf, cnt)
int fd;
char *buf;
int cnt;
```

This routine writes 'cnt' characters from the buffer
whose address is given by 'buf' to the file or device
specified by 'fd'. The actual number of characters
written is returned. A negative error number is
returned if an error occurs.

4.4 Utility Routines

The utility functions consist primarily of routines which deal with memory allocation and manipulation and string functions. Many of these routines are written in 6502 assembly language to improve performance.

alloc   C

        char *alloc(size)
        int size;

        Allocates memory with size number of bytes and returns pointer to beginning. Returns NULL if too close to stack. Memory allocated by alloc() cannot be released using free().

atof   6502 Assembly

        double atof(str)
        char *str;

        Convert the null terminated ASCII string pointed to by str into a floating point number.

atoi   C

        atoi(str)
        char *str;

        Converts a string of decimal digits into a signed integer value. Conversion stops on the first non-numeric value encountered in the string.

atol   C

        long atol(str)
        char *str;

        Converts a string of decimal digits into a signed long value. Conversion stops on the first non-numeric value encountered in the string.

blockmv   6502 Assembly

        blockmv(dest, src, cnt)
        char *dest, *src;
        int cnt;

        This routine moves cnt bytes from dest to src. The routine checks for overlap and starts at the appropriate end.

calloc   C

```
char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

Calls malloc asking for (nelem*elsize) bytes of memory.
Included for completeness.

clear   6502 Assembly

```
clear(start, len, fillc)
char *start;
int len;
char fillc;
```

Fills the len bytes starting at start with the
character fillc.

execl   C

```
execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;
```

Overlays the calling program with the program specified
by "name". There is no return to the calling program.
File descriptors remain open across the call.
Conventionally, "arg0" is the same as the name of the
program being called. The argument list is terminated
by a null argument. This routine is only available
under the SHELL, and is not supported by Apple DOS.

format   C

```
format(function, format, argptr)
int (*function)();
char *format;
unsigned *argptr;
```

Formats data according to the string, "format", and
calles the given function with each character of the
result. Formatting is done as described in Chapter 7,
Input and Output, of The C Programming Language.

e.g. the printf() routine looks like this:

```
printf(fmt, args)
char *fmt;
unsigned args;
{
    int putchar();

    format(putchar, fmt, &args);
}
```

free   C

        free(addr)
        char *addr;

        Deallocates a previously malloc'ed or calloc'ed piece
        of memory. Adjacent free spaces are collapsed.

ftoa   6502 Assembly

        ftoa(flt, str, precision, type)
        double flt;
        char *str;
        int precision, type;

        Convert from float/double format to an ASCII string.
        The value of flt is converted to ASCII and stored in
        the space pointed to by str. Precision specifies the
        number of digits to the right of the decimal point. If
        type is 0, the format will be of the printf F type. If
        type is 1, the string will be in E format. This is the
        routine used by format.

htoi   C

        htoi(str)
        char *str;

        Converts a string of hex digits into an unsigned
        integer value. Conversion stops on the first non-hex
        value encountered in the string.

index   6502 Assembly

        char *index(str, c)
        char *str;
        char c;

        Returns a pointer to the first occurence of character c
        in the null-terminated string str. Returns 0 if c is
        not found.

isdigit   C

        isdigit(c)
        int c;

        Returns one if c is a digit, zero otherwise.

islower   C

```
islower(c)
int c;
```

Returns one if c is a lower case alphabetic, zero otherwise.

isspace   C

```
isspace(c)
int c;
```

Returns one if c is space, tab, carriage return, newline or form feed, zero otherwise.

isupper   C

```
isupper(c)
int c;
```

Returns one if c is an upper case alphabetic, zero otherwise.

malloc   C

```
char *malloc(size)
int size;
```

Returns a pointer to size bytes. This space is allocated starting at the end of the current program. A map is kept of allocated space. The companion routine free(), returns space to the usable pool. If a large enough piece of memodry is not available from the pool, the top of allocated memory is remembered and each call adds the size requested to the top value. If the top value is within 256 bytes of the bottom of the stack, NULL is returned. Note, that the amount of space allocated is actually two bytes larger to store the size.

rindex   6502 Assembly

```
char *rindex(str, c)
char *str;
char c;
```

Returns a pointer to the last occurence of character c in the null-terminated string str. Returns 0 if c is not found.

rwts   6502 Assembly

```
rwts(track, sector, buf, cmd, slot, drive, vol)
int track, sector;
char *buf;
int cmd, slot, drive, vol;
```

Sets up the RWTS param block according to the arguments
and then calls the RWTS routine. Returns 0 if no error,
else the negative of the error number. For more
information consult the Apple DOS manual.

settop   C

```
char *settop(size)
int size;
```

The current top of available memory is moved up by size
bytes and the old value of the top is returned. If the
new value is within 256 bytes of the stack pointer,
NULL will be returned. The initial top of memory is set
by the program initialization code to the end of the
program.

strcat   6502 Assembly

```
char *strcat(s1, s2)
char *s1, *s2;
```

Appends a copy of string s2 to the end of string s1. A
pointer to the null terminated result is returned.

strcmp   6502 Assembly

```
strcmp(s1, s2)
char *s1, *s2;
```

Compares its arguments character by character and
returns an integer greater than, equal to, or less than
0, according as s1 is lexicographically greater than,
equal to, or less than s2.

strcpy   6502 Assembly

```
strcpy(s1, s2)
char *s1, *s2;
```

Copies string s2 to string s1 including the null
character. Returns s1.

strlen   6502 Assembly

```
strlen(str)
char *str;
```

Returns the length of the string str (not including the
terminating null character.)

strncat   6502 Assembly

```
char *strncat(s1, s2, n)
char *s1, *s2;
```

Appends at most n characters of string s2 to the end of
string s1. A pointer to the null terminated result is
returned.

strncmp   6502 Assembly

```
strncmp(s1, s2, n)
char *s1, *s2;
```

Same comparison as strcmp, but compares only up to n
characters.

strncpy   6502 Assembly

```
strncpy(s1, s2, n)
char *s1, *s2;
```

Copies at most n characters of string s2 to string s1.
If a null character is encountered before the n'th
character, s1 will be padded with nulls. If n
characters are copied and a null character has not been
copied no null is placed in s1. Returns s1.

system   C

```
system(str)
char *str;
```

The string, "str", is passed to and interpreted by the
SHELL as a command. This routine is only available
through the SHELL and is not supported by Apple DOS.

tolower   C

```
tolower(c)
char c;
```

Returns the character c in lower case if it is a lower
case alphabetic, otherwise c is returned unchanged.

toupper   C

```
toupper(c)
char c;
```

Returns the character c in upper case if it is a lower
case alphabetic, otherwise c is returned unchanged.

4.6 Math Routines

acos

```
double acos(x);
double x;
```

acos is a function of one argument which, when called, returns as its value the arcosine of the argument. The returned value is of type double.

The absolute value of the argument must be less than or equal to 1.0. It must be of type double.

If acos can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. The accompanying table lists the symbolic codes which acos may set in ERRNO, the associated value returned by acos, and the meaning.

Error codes returned in ERRNO by acos

Code	acos (x)	Meaning
EDOM	0.0	abs(x) > 1.0

asin

```
double asin(x);
double x;
```

asin is a function of one argument which, when called, returns as its value the arcsine of the argument. The returned value is of type double.

The absolute value of the argument must be less than or equal to 1.0. Its type is double.

If asin can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. The accompanying table lists the symbolic codes which asin may set in ERRNO, the associated values returned by asin, and the meaning.

Error codes returned in ERRNO by asin

Code	asin(x)	Meaning
EDOM	0.0	abs(x) > 1.0

atan

```
double atan(x);
double x;
```

atan is a function of one argument which, when called, returns as its value the arctangent of the argument. The returned value is of type double.

The argument can be any real value, and must be of type double.

Unlike many of the other math functions, atan never returns code in ERRNO.

atan2

```
double atan2(y,x);
double y,x;
```

atan2 is a function of two arguments, say x and y, which, when called, returns as its value the arctangent of y/x, in radians. y is the first argument, and x is the second. The returned value is of type double.

The arguments can assume any real values, except that x and y cannot both be zero. If x equals zero, the value returned is also zero.

If atan2 can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. The accompanying table lists the symbolic codes which atan2 may set in ERRNO, the associated values returned by atan2, and the meaning.

Error codes returned in ERRNO by atan2

Code	atan2(x)	Meaning
EDOM	0.0	x = y = 0

cos

```
double cos(x);
double x;
```

cos is a function of one argument which, when called, returns as its value the cosine of the argument. The returned value is of type double.

The argument is in radians, and its absolute value must be less than 6.7465e9. The type of the argument is double.

If cos can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value, without modifying the associated value returned by cos, and the meaning.

Error codes returned in ERRNO by cos

Code	cos(x)	Meaning
ERANGE	0.0	abs(x) >= 6.7465e9

cosh

```
double cosh(x);
double x;
```

cosh is a function of one argument which returns as its value the hyperbolic cosine of the argument. The value returned is of type double.

The absolute value of the argument must be less than 348.606839, and it must be of type double.

If cosh can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. The accompanying table lists the symbolic codes which cosh may set in ERRNO, the associated values returned by cosh, and the meaning.

Error codes returned in ERRNO by cosh

Code	cosh(x)	Meaning
ERANGE	5.2e151	abs(x) > 348.606839

cotan

```
double cotan(x);
double x;
```

cotan is a function of one argument which, when called, returns as its value the cotangent of the argument. The returned value is of type double.

The argument is in radians, and its absolute value must be greater than 1.91e-152 and less than 6.7465e9. The type of the argument is double.

If cotan can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. The accompanying table lists the symbolic codes which cotan may set in ERRNO, the associated value returned by cotan, and the meaning.

Error codes returned in ERRNO by cotan

Code	cotan(x)	Meaning
ERANGE	5.2e151	0<x<1.91e-152
ERANGE	-5.2e151	-1.91e-152 <x<0
ERANGE	0.0	abs(x) >= 6.7465e9

exp

```
double exp(x);
double x;
```

exp is a function of one argument which returns as its value e**(argument). The type of the returned value is double.

The argument must be greater than -354.8 and less than 349.3; it must be of type double.

If exp is unable to perform the computation, it sets a code in the global integer ERRNO and returns an arbitrary value; otherwise, it returns the computed value without modifying ERRNO. The accompanying table lists the symbolic values that exp may set in ERRNO, the associated value of exp, and the meaning.

Error codes returned in ERRNO by exp

Code	exp(x)	Meaning
ERANGE	5.2e151	x > 349.3
ERANGE	0.0	x < -354.8

log

```
double log(x);
double x;
```

log is a function of one argument which returns the natural logarithm of the argument as its value, as a double precision floating point number.

The argument which is passed to log must be a double precision floating point number and must be greater than zero.

If log detects an error, it sets a code in the global variable ERRNO and returns an arbitrary value;otherwise, it returns to the caller without modifying ERRNO. The accompanying table lists the symbolic values which log may set in ERRNO, the associated values returned by log, and the meaning.

Error codes returned in ERRNO by log

Code	log(x)	Meaning
EDOM	-HUGE	x <= 0.0

log10

```
double log10(x);
double x;
```

log10 is a function of one argument which returns as its value the base-10 logarithm of the argument. The type of the returned value is double.

The argument must be greater than zero, and must be of type double.

If log10 detects an error, it sets a code in the global integer ERRNO and returns an arbitrary value to the caller; otherwise, it returns to the caller without modifying ERRNO. The accompanying table the symbolic values which log10 may set in ERRNO, the associated value returned by log10, and the meaning.

Error codes returned in ERRNO by log10

Code	log10(x)	Meaning
EDOM	-5.2e151	x <= 0.0

pow

```
double pow(x,y);
double x,y;
```

pow is a function of two arguments, for example, x and
y, which, when called, returns as its value x to the y-
th power (x**y, in FORTRAN notation). x is the first
argument to pow, and y the second. The value returned
is of type double.

The arguments must meet the following requirements:
      x cannot be less than zero;
      if x equals zero, y must be greater than zero;
      if x is greater than zero, then
            $-354.8 < y*\log(x) < 349.3$

If pow is unable to perform the calculation, it sets a
code in the global integer ERRNO and returns an
arbitrary value; otherwise it returns the computed
number as its value without modifying ERRNO. The
accompanying table lists the symbolic codes which pow
may set in ERRNO, the associated value returned by pow,
and the meaning.

Error codes returned in ERRNO by pow

Code	pow(x,y)	Meaning
EDOM	-5.2e151	x<0 or x=y=0
ERANGE	5.2e151	y*log(x) > 349.3
ERANGE	0.0	y*log(x) < -354.8

random

```
double random()
```

returns a random number in the range 0 to 1.

sin

```
double sin(x);
double x;
```

sin is a function of one argument which, when called,
returns as its value the sine of the argument. The
value returned is of type double.

The argument is in radians, and its absolute value must
be less than 6.7465e9. The type of the argument is
double.

If sin can't perform the computation, it returns an
arbitrary value and sets a code in the global integer
ERRNO; otherwise, it returns the computed number,
without modifying ERRNO. The accompanying table lists
the symbolic codes which sin may set in ERRNO, the
associated values returned by sin, and the meaning.

Error codes returned in ERRNO by sin

Code	sin(x)	Meaning
ERANGE	0.0	abs(x) >= 6.7465e9

sinh

```
double sinh(x);
double x;
```

sinh is a function of one argument which returns as its
value the hyperbolic sine of the argument. The returned
value is of type double.

The absolute value of the argument must be less than
348.606839, and is of type double.

If sinh can't perform the computation, it sets a code
in the global integer ERRNO and returns an arbitrary
value; otherwise, it returns the computed value without
modifying ERRNO. The accompanying table lists the
symbolic codes which sinh may set in ERRNO, the value
returned by sinh, and the meaning.

Error codes returned in ERRNO by sinh

Code	sinh(x)	Meaning
ERANGE	5.2e151	abs(x) > 348.606839

sqrt

```
double sqrt(x);
double x;
```

sqrt is a function of one argument which returns as its value the square root of the argument. The type of the returned value is double.

The argument which is passed to sqrt must be of type double and must be greater than or equal to zero.

If sqrt detects an error, it sets a code in the global integer variable ERRNO and returns an arbitrary value to the caller. If sqrt doesn't detect an error, it returns to the caller without modifying ERRNO. The accompanying table lists the symbolic values which sqrt may set in ERRNO and their meanings. The file MATH.H, which can be included in a user's module, declares ERRNO to be a global integer and defines the numeric value associated with each symbolic value.

Error codes returned in ERRNO by sqrt

Code	sqrt(x)	Meaning
EDOM	0.0	x < 0.0

tan

```
double tan(x);
double x;
```

tan is a function of one argument which, when called, returns as its value the tangent of the argument. The type of the value returned is double.

The argument is in radians, and its absolute value must be less than 6.7465e9. The type of the argument is double.

If tan can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. The accompanying table lists the codes which tan may set in ERRNO, the associated value returned by tan, and the meaning.

Error codes returned in ERRNO by tan

Code	tan(x)	Meaning
ERANGE	0.0	abs(x) >= 6.7465e9

Technical Information

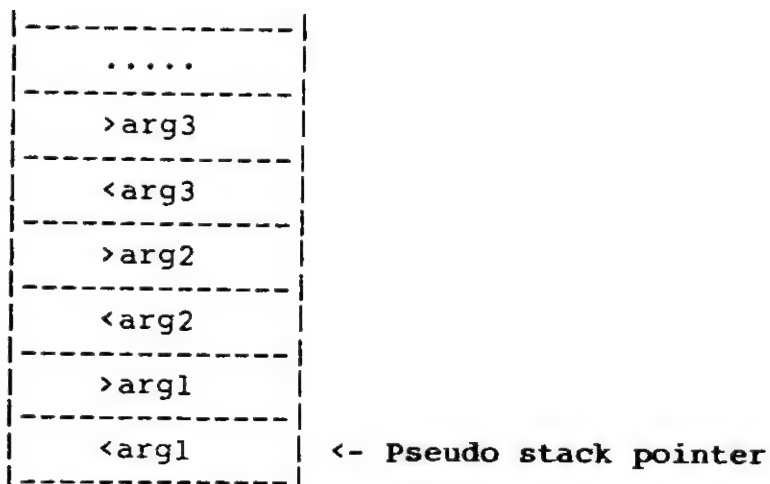# SECTION 5 - TECHNICAL INFORMATION

5.1 Introduction

This section of the manual contains information on a variety
of different subjects. Each section is independent of the others
and are in no particular order. Other sections containing
additional information will undoubtedly be added in the future.

5.2   Interfacing to Assembly Language

The calling conventions used by both the C65 and CCI compilers are fairly simple and identical. Both compilers produce code which makes use of a pseudo-stack which is kept in locations 2-3 of memory. This stack is used for passing arguments to the routine being called. The arguments are pushed onto the stack in reverse order, i.e. the last argument is pushed first and the first argument is pushed last. The following example illustrates this point.

In this example, the function is called with three arguments. The diagram is a pictorial representation of the pseudo-stack when the function is called.

func(arg1, arg2, arg3);

```
|---------------|
|     .....     |
|---------------|
|    >arg3      |
|---------------|
|    <arg3      |
|---------------|
|    >arg2      |
|---------------|
|    <arg2      |
|---------------|
|    >arg1      |
|---------------|
|    <arg1      |   <- Pseudo stack pointer
|---------------|
```

The return address is on the 6502 machine stack and can be left there, if desired, till the function returns. The code generated by the compiler moves the return address to the pseudo-stack allowing fully re-entrant code without the limitation of the 6502 machine stack.

Accessing the arguments is fairly simple as illustrated in the following short assembly language program which adds "arg1" to "arg3" and returns the value to the calling function.

```
SP          equ        2
R0          equ        8
*
            public     test_
test_       clc
            ldy        #0
            lda        (SP),Y
            ldy        #4
            adc        (SP),Y
            sta        R0
            ldy        #1
            lda        (SP),Y
            ldy        #5
            adc        (SP),Y
            sta        R0+1
            rts
```

There are several lessons which can be learned from this simple example. First, the C compiler adds an '_' character to the end of all symbol names. However, this is only true if the name is less than 8 characters. Second, the name of the function to be called by a C routine must be declared public, or the linker will not connect references in a different module with the label defined in this one.

Notice also that the result of the function is returned in locations 8-9. This is true for both compilers. In actuality, locations 8-B constitute a 32-bit register. If a function is declared to return a long value, all four locations are used. For char, int, unsigned int, or pointer only the first two locations are used.

Locations 0-1F are defined as follows:

Location	Use
0- 1	Temporary storage
2- 3	Pseudo stack pointer
4- 5	Pseudo frame pointer
6- 7	Pseudo program counter/frame pointer 2
8- B	Register R0
C- F	Register R1
10-13	Register R2
14-15	Pointer to floating point accumulator
16-17	Pointer to floating point secondary
18-1b	Register R3
1c-1f	Register R4

An assembly language function may make use of any of the pseudo-register locations. However, the data in locations 2-7 must be preserved and intact when the function returns. If floating point is being used, the pointers to the floating point

registers must also be preserved. If a C function is called from assembly language, any values in the pseudo-registers may be destroyed by the C routine.

There are several topics concerning the linker which are important if the assembler and linker are to be used without any compiled code. The linker automatically creates two symbols in its symbol table, "MEMRY" and ".begin". The symbol "MEMRY" is defined by the linker to be a label placed at the end of the code and data segment of a program. It is used to set the top of memory for the memory allocation routines. It is used as an address. The values stored at that address are not defined as they are beyond the end of the program.

The second symbol, ".begin", is used to force loading of the startup routine from the library. This symbol is set as undefined in the linker's symbol table. If a program is linked with the library, this symbol will be defined in the "crt0.rel" file. If a program is written completely in assembly language, and linked with the linker, the ".begin" label must be defined somewhere in the assembly language program as follows:

```
          public    .begin
.begin
```

which will satisfy the linker.

As a final note, the C65 compiler does support the #asm-#endasm control structure. This allows 6502 assembly language to be directly embedded within a C routine. Finding a good example where this construct is necessary is very difficult.

```
rotate(arg)
{
     register int i;

     i = arg;
#asm
     lda  $81
     rol  A
     rol  $80
     rol  $81
#endasm
     return(i);
}
```

This routine rotates a two byte quantity one bit to the left. This operation is messy in C and in a time critical application not feasible to make an assembly language subroutine. This routine is not a good example, since it would be better to write the entire thing in assembly. However, in the middle of a larger routine, it might conceivably be useful. This facility is provided as a last resort and is generally not recommended as it is completely non-portable.

5.3 ROMable Code

Aztec C65 produces reentrant code that is ROMable. The basic tools for producing ROMable code are provided by the linker, LN. However, there are a number of restrictions and support routines which need to be supplied by the user. The following discussion will highlight these.

Programs can be visualized as having three major components, the code, the initialized data, and uninitialized data. The code consists of the compiled program itself. The compiler generates the "cseg" pseudo-op before each section of compiled code to differentiate it from data. String constants are placed in the code segment as well.

With the current linker, initialized and uninitialized data are grouped together using either the "dseg" or "common" pseudo-ops. The data is normally placed immediately after the code segment.

To place a program in ROM, the "-c" option must be used to specify the beginning address of the code section of the program. This should correspond to the final address of the ROM in memory. Likewise, the "-d" option must be used to specify the location of the data section of the program, which in this case will be the address of the available RAM in the system. The "-b" option specifies the base address of the output file and should be set to the lesser of the code and data addresses.

When the modules are linked, both code and data will be placed in the output file with nulls filling the space between. Separating the code and data is up to the user. The size of each is displayed by the linker.

The startup routine supplied with the libraries is almost certainly unsuitable for a ROM application and must be rewritten. This routine sets up the stack pointer and the beginning of available memory for the allocation routines. It must be modified to reflect the configuration of the machine being used. In addition, if there is any initialized data, that data must either be initialized directly by the startup routine, or a copy of the data can be kept in ROM and copied to RAM as part of the startup routine.

The startup routine then calls the croot() routine. The croot() routine initializes the standard in, out and error I/O links and calls main(). This routine may also have to be modified to suit the specific application.

5.4   Differences Between the SHELL and Stand-alone Libraries

The stand-alone library is identical to the libraries used with the SHELL except for three areas. First, the concept of an execution drive is not supported in the stand-alone library. All open() calls that do not include a specific drive number as part of the name will search the last disk accessed through Apple DOS.

Second, the execl() and system() calls are functions which are built into the SHELL, and which cannot be easily duplicated with Apple DOS directly. Likewise, the concepts of arguments to a program and I/O redirection as part of the program invocation are just not available under Apple DOS. However, these facilities can be added as part of the program startup. The croot() routine in particular can be modified to prompt for a command line which can then be parsed for arguments and for I/O redirection commands. The library supplied does not provide these modifications.

Finally, the character device I/O is substantially different from that supported by the SHELL. Whereas the SHELL device drivers support a variety of keyboards, screens and printers, the stand-alone drivers only support the zero-page input and output vectors, KSW and CSW. There are two reasons for this difference.

First, the device drivers in the SHELL reside in Bank 1 of the extended RAM card. This memory is normally unused by just about everybody. As such, there is 4k free for building sophisticated general purpose device drivers. When a program is linked with the stand-alone library, everything must fit in the space from $800 to the bottom of the DOS buffers. The overhead of a 3-4k device driver may not be acceptable.

Secondly, there is the fact that the driver will have to be configured as well. Most programs linked with the stand-alone library are written with the intent of running them on other machines. It is unlikely that the configuration of a different machine will match that of the development machine. By using the KSW and CSW vectors, 80-column screens, printers, modems, and other peripherals can be accessed by using the Apple I/O redirection commands, "IN#" and "PR#".

Otherwise, the two libraries contain the exact same functions and may be used in precisely the same way.

5.5 Using the Stand-alone Library

The stand-alone library is used in much the same manner as the SHELL library. When the program is started, the startup code determines whether Applesoft or Integer Basic is active and sets the pseudo-stack pointer to the appropriate HIMEM value. The croot() routine then initializes the first three file descriptors to the first entry in the character device table.

As supplied, the character device table only contains the device name, "KB:", which is used for both the keyboard and screen. By initializing the file descriptor table directly, and not using the open() routine, the system I/O routines are not included by the linker, decreasing the default program size.

The croot() routine then calls the main() routine with an argument count of zero, since it is not possible to specify arguments through Apple DOS. When the main() routine returns, or the exit() routine is called, control is returned to Apple DOS by jumping to the DOS warm start vector at $3D0.

5.6 Writing Small Programs

Program size can be divided into two factors. The size of the actual program and the size of the library routines which are added by the linker. The first factor can be affected by good programming technique, use of subroutines and allocation of register variables.

The second factor deserves a more detailed discussion. The Aztec linker, LN, will only include a module from the library which satisfies an undefined symbol from a previous module. If a module contains more than one function, the linker can only include the whole module, even if the second function is not used by the program being linked. For this reason, most library functions are contained in separate modules.

Because the linker will add undefined symbols from a module early in the library to its list of undefined symbols, including one module may result in the inclusion of several more. This has the most significance for the standard I/O library and the system I/O library.

Any I/O which is done using the library routines is eventually done through the system I/O routines. If a creat() or open() call is made from a function, routines for searching the DOS buffers, parsing the drive, slot and volume number and searching the device name table, will all be linked into the program. If a program does not perform any I/O, these routines will not be included from the library.

Since a program which does no I/O is fairly rare, most programs will at least include the basic system I/O routines. Most of these routines are used for doing file I/O. If no disk I/O is necessary, then the best way to write a small program is to write two small assembly language routines to get and put characters from and to the KSW and CSW zero-page vectors. These routines can be patterned after the routines which are included in the stand-alone library source archives.

The really big kicker is the standard I/O library. There are several drawbacks to the standard I/O library for writing small programs. First, it does not do the I/O directly, and thus calls the system I/O routines to perform the operation. This means that the code for the standard and the system I/O libraries must be included. This also means that the processing performed on each character is significant, since four or five routines must be called for each character. The advantage of standard I/O is portability, but it doesn't come for free.

5.7   Overlay Support

5.7.1 Overview

Applications often require more memory than the limited memory of a microcomputer. Overlays are one solution to this problem. They allow a user to divide a program into several segments. One of the segments, called the root segment, is always in memory. The other segments, called overlays, reside on disk and are only brought into memory when requested by the root segment.

If an overlay is in the overlay area of memory when the root requests that another be loaded, the newly specified overlay segment overlays the first. There is no limit to the number of overlays that can be called.

Nothing special must be done to functions which reside in an overlay. The overlay doesn't have to know that it's an overlay. An overlay is activated in the normal way, and performs a normal return.

Several files are created in the process of building an overlaid program. First, there is a file containing the relocatable symbol table with the extent ".rsm" for the root and for any overlay that invokes another overlay. Second, there is one file for each overlay with the extent ".ovr". Finally, the root segment appears as a normal binary program with no extension.

The following is a sample set of commands to create a root with two overlays.

```
ln -r myroot.rel ovloader.rel sh65.lib
ln mysub1.rel myroot.rsm ovbgn.rel sh65.lib
ln mysub2.rel myroot.rsm ovbgn.rel sh65.lib
```

The "-r" option in the first command denotes that the module being created is a root. Files "myroot.rsm" and "myroot" are created. The "-r" option is not specified in the second command since it does not invoke another overlay. The presence of the ".rsm" file in the parameter list is all that is necessary to inform the linker that "mysub1.rel" is an overlay file. The third command creates a second overlay file called "mysub2.ovr" that runs in the same space as the first overlay file.
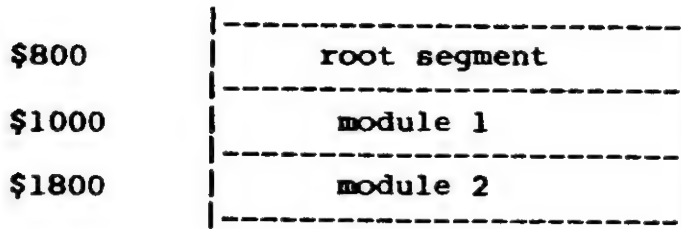
address

```
           |-------------------------|
    $800   |       root segment      |
           |-------------------------|
   $1000   |        module 1         |
           |-------------------------|
   $1800   |       module 2          |
           |-------------------------|
```

Figure 1

```
           |-------------------------|
    $800   |       root segment      |
           |-------------------------|
    $1000  |         |       |      $1000
    |-------------------|  |--------------------|
    |     module 1      |  |      module 2      |
    |-------------------|  |--------------------|
```
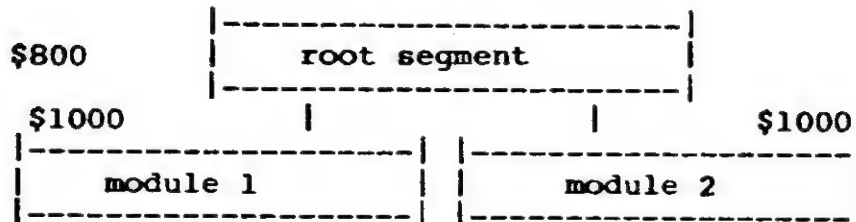
Figure 2

Figure 1 shows a program that can be logically divided into three segments as it would look if run as a single module. Figure 2 shows the same program run as an overlay. In figure 2, module 1 and module 2 occupy the same memory locations. A possible flow of control would be for the base routine to call module 1, module 1 then returns to the root and the root calls module 2, module 2 returns to the root and the root calls module 1 again. Then module 1 returns to the root and the root exits to the operating system. Notice that all overlay segments must return to their caller and that overlays at the same level cannot directly invoke each other.

5.7.2 Programmer Information

The root causes an overlay to be loaded into memory and control to be passed to it by calling the MANX-supplied function "ovloader", which must reside in the root segment. The parameters to ovloader are a character string, giving the name of the overlay to be loaded, followed by the parameters which are to be passed to the overlay. "ovloader" is of type "int".

When the overlay is loaded, control passes to the MANX-supplied function "ovbgn". In turn, ovbgn transfers control to the function "ovmain()" which must be defined in each overlay module. The stack is modified by "ovbgn" to make it appear as though the routine which called "ovloader()" called "ovmain()" directly. Thus, the return and return value transfer directly to the calling routine.

5.7.3 Example

In this example, the root segment, which consists of the function "main" and any necessary run-time library routines, behaves as follows:

 (1) it calls the overlay ovly1, passing it as parameter a
   pointer to the string "first message".
 (2) it prints the integer value returned to it by ovly1.
 (3) it calls the overlay ovly2, passing it a pointer to the
   string "second message".
 (4) it prints the integer value returned to it by ovly2.

The overlay segment ovly1 consists of the function ovmain(), the MANX function ovbgn, and any necessary run-time library routines not already in the root. It prints the message "in ovly1" plus whatever character string was passed to it by main().

The overlay segment ovly2 consists of the function ovmain(), the MANX function ovbgn, and any necessary run-time library routines not already in the root. It prints the message "in ovly2" plus whatever character string was passed to it by main().

Here then is the main function:

```
main()
{
     int a;

     a = ovloader("ovly1", "first message");
     printf("in main. ovly1 returned %d\n", a);
     a = ovloader("ovly2", "second message");
     printf("in main. ovly2 returned %d\n", a);
}
```

Here is ovly1:

```
ovmain(dummy, a)
char *a;
{
     printf("in ovly1. %s\n", a);
     return(1);
}
```

Here is ovly1:

```
ovmain(dummy, a)
char *a;
{
     printf("in ovly2. %s\n", a);
     return(2);
}
```

To create the appropriate pieces:

```
ln -r main.rel ovloader.rel sh65.lib
ln ovly1.rel ovbgn.rel main.rsm sh65.lib
ln ovly2.rel ovbgn.rel main.rsm sh65.lib
```

and when the program is run:

```
main
in ovly1. first message
in main. ovly1 returned 1.
in ovly2. second message
in main. ovly2 returned 2.
```

5.7.4 Caveats

There are several caveats if you are using overlays. First, the "settop()" function must be called with an argument whose value is equal to the size of the largest overlay segment, or the length of the longest thread if overlays are nested. This call must be initiated at the beginning of the main routine before calling any other routines. The size of an overlay is displayed on the console, in hex, when it's linked.

For example, if your program uses three overlays, and the linker says their sizes are $125A, $236, and $837, then it should make the following call:

```
settop(0x125a);
```

The parameter to settop could be larger, if desired.

Second, if standard I/O is used in an overlay segment at least one standard I/O call must exist in the root. This can be that it is not necessary that the call to "putc()" be executed, only that it be present. This will insure that the linker will include the standard I/O tables in the root segment.

5.7.5 Nesting Overlays

If one overlay is to call another, the command line to the linker for the first overlay must include the "-r" option. This will cause LN to generate a ".rsm" file for the first overlay. This ".rsm" file must then be included in the command line to the nested overlay. Also, each overlay must include the module "ovbgn.rel".

Overlays can be nested to any level. The "-r" option need not be specified for the last segment of any one overlay path. Extreme caution should be used in using an overlay segment in more than one path. Although this is possible to do, there is an enormous potential for error.

5.8   Data Formats

5.8.1 character

Characters are 8 bit ASCII.

Strings are terminated by a NULL (X'00').

For computation characters are promoted to integers with a value range from 0 to 255.

5.8.2 pointer

Pointers are two bytes (16 bits) long. The internal representation of the address F0AB stored in location 100 would be:

location   contents in hex format

   100   AB
   101   F0

5.8.3 int, short

Integers are two bytes long. A negative value is stored in two's compliment format. A -2 stored at location 100 would look like:

location   contents in hex format

100   FE
101   FF

5.8.4 long

Long integers occupy four bytes. Negative values are stored in two's complement representation. Longs are stored sequentially with the least significant byte stored at the lowest memory addres and the most significant byte at the highest memory address.

5.8.5 float and double

Floating point numbers are stored as 32 bits, doubles are stored as 64 bits. The first bit is a sign bit. The next 7 bits are the exponent in excess 64 notation. The base for the exponent is 256. The remaining bytes are the fractional data stored in byte normalized format. A zero is a special case and is all 0 bits. The hexadecimal representation for a float and double with a value of 1.0 is:

      41 01 00 00   41 01 00 00 00 00 00 00

A 255.0 would be:

      41 FF 00 00   41 FF 00 00 00 00 00 00

A 256.0 would be:

      42 01 00 00   42 01 00 00 00 00 00 00

5.9   Floating Point Support

5.9.1   Overview

Aztec C65 supports floating point numbers of type float and double. All arithmetic operations (add, subtract, multiply, and divide) can be performed on floating point numbers, and conversions can be made from floating point representation to other other representations and vice versa.

The common conversions are performed automatically, as specified in the K & R text. For example, automatic conversion occurs when a variable of type 'float' is assigned to a variable of type 'int', or when a variable of type 'int' is assigned to a variable of type 'float', or when a 'float' variable is added to an 'int' variable.

Other conversions can be expicitly requested, either by using a 'cast' operator or by calling a function to perform the conversion. For example, if a function expects to be passed a value of type 'int', the (int) cast operator can be used to convert a variable of type 'float' to a value of type 'int', which is then passed to the function. For another example, the function 'atof' can be called to convert a character string to a value of type 'double'.

The following sections provide more detailed information of the floating point system. One section describes the internal representation of floating point numbers and another describes the handling of exceptional conditions by the floating point system.

5.9.2   Floating point exceptions

When a C program requests that a floating point arithmetic operation be performed, a call will be made to functions in the floating point support software. While performing the operation, these functions check for the occurence of the floating point exception conditions; namely, overflow, underflow, and division by zero.

On return to the caller, the global integer 'flterr' indicates whether an exception has occurred. If the value of this integer is zero, no error occurred, and the value returned is the computed value of the operation. Otherwise, an error has occurred, and the value returned is arbitrary. Table A lists the possible settings of flterr, and for each setting, the associated value returned and the meaning.

flterr	value returned	meaning
0	computed value	no error has occurred
1	+/- 2.9e-157	underflow
2	+/- 5.2e151	overflow
3	+/- 5.2e151	division by zero

When a floating point exception occurs, in addition to returning an indicator in 'flterr', the floating point support routines will log an error message to the console. The error message logged by the support routines define the type of error that has occurred (overflow, underflow, or division by zero) and the address, in hex, of the instruction in the user's program which follows the call to the support routines.

Following the error-message-logging the floating point support routines return to the user's program which called the support routines.

To determine whether to log an error message itself or to call a user's function, the support routines check the first pointer in Sysvec[], the global array of function pointers. If it contains zero (which it will, unless the user's program explicitly sets it), the support routines log a message; otherwise, the support routines call the function pointed at by this field.

A user's function for handling floating point exceptions can be written in C. The function can be of any type, since the support routines don't use the value returned by the user's function. The function has two parameters: the first, which is of type "int", is a code identifying the type of exception which has occurred. One indicates underflow, two overflow, and three division by zero.

The second parameter passed to the user's exception-handling routine is a pointer to the instruction in the user's program which follows the call instruction to the floating point support routines. One way to use this parameter would be to declare it to be of type "int". The user's function could then convert it to a character string for printing in an error message.

5.9.3 Example

Two programs follow. One is a sample routine for handling floating point, followed by exceptions. The routine displays an error message, based on the type of error that has occurred. The other is main(), which sets a pointer to the error-handling routine in the Sysvec[] array.

```
#include "math.h"

main()
{
    Sysvec[FLT_FAULT] = usertrap;
}

usertrap(err, addr)
int err;
char *addr;
{
    switch(err) {
    case 1:
        printf("floating point underflow at %04x\n", addr);
        break;
    case 2:
        printf("floating point overflow at %04x\n", addr);
        break;
    case 3:
        printf("floating pt divide by 0 at %04x\n", addr);
        break;
    default:
        printf("invalid code %d passed to usertrap\n", err);
        break;
    }
}
```

5.9.4 Internal Representation

Floats

A variable of type "float" is represented internally by a sign flag, a base-256 exponent in excess-64 notation, and a three-character, base-256 fraction. All variables are normalized.

The variable is stored in a sequence of four bytes. The most significant bit of byte 0 contains the sign flag; 0 means it's positive, 1 negative.

The remaining seven bits of byte 0 contain the excess-64 exponent.

Bytes 1, 2, and 3 contain the three-character mantissa, with the most significant character in byte 1 and the least in byte 3. The "decimal point" is to the left of the most significant byte.

As an example, the internal representation of decimal 1.0 is:

41 01 00 00

Doubles

A floating point number of type "double" is represented internally by a sign flag, a base-256 exponent in excess-64 notation, and a seven-character, base-256 fraction.

The variable is stored in a sequence of eight bytes. The most significant bit of byte 0 contains the sign flag; 0 means positive, 1 negative.

The excess-64 exponent is stored in the remaining seven bits of byte 0.

The seven-character, base-256 mantissa is stored in bytes 1 through 7, with the most significant character in byte 1, and the least in byte 7. The "decimal point" is to the left of the most significant character.

As an example, $(256**3)*(1/256 + 2/256**2)$ is represented by the following sequence of bytes:

43 01 02 00 00 00 00 00

5.10 Device Ioctl

5.10.1 Overview

Peripheral devices tend to have certain attributes associated with them. It is often useful to be able to modify or access these attributes in ways that are fairly uniform. The ioctl() call provides one such mechanism in the Aztec system for the Apple. Commands between 0 and 99 are reserved for definition by Manx Software Systems. The following sections detail the currently defined set of I/O controls for disk and character devices.

5.10.2 Disk Ioctl

These ioctl commands are used for devices which support the DOS file system.

CMD ARG	DEFINITION
0	Determine whether the device is file oriented. Returns -1.
1	Returns the address of the DOS buffer.
2	Returns the file type of the file.
3	Returns zero if file unlocked, else non-zero.
4-99	Undefined.

5.10.3 Character Device Ioctl

These ioctl commands are use for devices which are single character oriented devices. These definitions are available in the header file "kbctl.h".

In general, if a control is invalid, -1 will be returned. For the screen controls, a zero is returned if the function was available. For the mapping controls, the previous setting is returned. For example, if interrupts are to be ignored, the call would be:

        i = ioctl(fd, KB_INTR, 0);

The value in "i" would be 0 if interrupts were already off, else it would be 1 if they were on.

CMD	ARG		DEFINITION
KB_STAT	0	0/1	Returns 0 if character device.
KB_ECHO	1	0/1	Controls automatic keyboard echo.
KB_IMAP	2	0/1	Controls input mapping of control codes.
KB_INTR	3	0/1	Controls ^C interrupt.
KB_STOP	4	0/1	Controls ^S output control.
KB_CRLF	5	0/1	Controls mapping of CR to CR/LF.
KB_TAB	9	N	If 0, returns tab width, else sets to N.
KB_OMAP	10	0/1	Controls output table mapping.
KB_WID	11		Returns the width of the output device.
KB_LEN	12		Returns the length of the output device.
KB_CURS	13	X.Y	Move the cursor to position X, Y. (See note)
KB_CLEAR	14		Home cursor and clear screen.
KB_CLEOS	15		Clear to end of screen from cursor.
KB_CLEOL	16		Clear to end of line from cursor.
KB_LINS	17		Insert a blank line at the cursor.
KB_LDEL	18		Delete the line at the cursor.
KB_CHINS	19		Insert a blank space at the cursor.
KB_CHDEL	20		Delete the character at the cursor.
KB_INV	21	0/1	Turns inverse off/on.
KB_SCRL	23		Scrolls screen up one line.
KB_RSCRL	24		Scrolls screen down one line.
KB_C_UP	25		Moves cursor up one line.
KB_C_RT	26		Moves cursor right one position.
KB_HOME	27		Moves cursor to the home position.

Not all the above listed functions are available for all devices. Refer to the CONFIG section of the SHELL reference section for more information.

NOTE: For cursor positioning, the X and Y coordinates are packed into two bytes, with the X coordinate being the high byte and the Y coordinate being the low byte. An example call would be:

        ioctl(fd, KB_CURS, (x << 8) | y);

5.11 Device Drivers

5.11.1 Overview

Device drivers are used to interface between programs and
input/output devices. These drivers may be written either in C or
in 6502 assembly language, or a combination of both. The two
objectives of this section are to explain how devices are
accessed from a C program and how new device drivers are written
and added to the system.

5.11.2 Access from C

Associated with each device is a name. This name is
arbitrary and may be any number of characters from 1 to 5. This
name must be known to access the device. As an illustration, the
name of the Apple keyboard and screen is "kb:". The key to using
devices is to treat them almost the same as files. The primary
difference is that there is no ability to seek within a non-file
device.

For example, to access a printer device whose name is "pr:",
first open the device for writing using either the open() or the
fopen() library routine. The file descriptor which is returned
can then be used in write() or putc() calls just as though it
were a file.

Associated with each device are five entry points. Not all
entry points need be available for each device. The five entry
points are "open", "close", "read", "write" and "ioctl". These
correspond to the system I/O routines of the same names. The
exact calling sequences are defined in the next section. If the
device does not have a table entry for the specified function, a
zero is returned and no action performed.

5.11.3 Internal Definition

A device driver module consists of one or more device
drivers linked together into a loadable binary image. The device
driver interface revolves around the device parameter table. This
table must be the very first element of the device driver module.
The module is always loaded into bank1 of the ram card at
location $D000.

The device parameter table is defined as follows:

```
struct device {
     char dev_name[6];
     int (*dev_open)();
     int (*dev_put)();
     int (*dev_get)();
     int (*dev_close)();
     int (*dev_ctl)();
} dev[8];
```

Dev_name is the name of the device from 1 to 5 characters, null terminated.

Dev_open is called when the device is opened. This might be used for initialization, e.g. form feed on a printer.

Dev_put is called when writes are done to the device. It is called with two arguments, an address of a buffer and a count. Dev_put must exist if the file is opened for writing.

Dev_get is called when reads are done from the device. It is called with two arguments, an address of a buffer and a count of the number of characters desired. The actual number of characters read is returned. If the device is opened for reading, an error is returned if dev_get is 0.

Dev_close is called when the device is closed. It is used to clean up if necessary.

Dev_ctl corresponds to the ioctl() routine call on the specified device. Dev_ctl is passed two arguments, a command and an argument. Commands between 0 and 99 may only be defined by Manx Software Systems in order to provide a standard set of control routines. The Device Ioctl section details the currently defined set of I/O controls.

Following the eight device table entries, at $D080, is the parameter table used by the SHELL device drivers to adapt to different configurations of hardware. The locations and their values are described in the following table:

ADDRESS	CONTENTS
D080	address of input routine
D082	address of output routine
D084	address of terminal characteristics array
D086	keyboard input flags
D087	screen output flags
D088	tab width
D089	screen width
D08A	screen length
D08B	quit character
D08C	CAPS LOCK character
D08D	stop output character
D08E	EOF character from keyboard

D08F	printer output flags
D090	length of printer initialization string
D091	address of printer initialization string
D093	address of keyboard input map table

The keyboard flags are defined as follows:

Bit	Meaning
0	echo characters on input if set
1	map carriage to newline if set
2	enable ^C break if set
3	enable ^S output stop if set
4	enable CAPS LOCK if set
5	enable shift wire mod if set
6	enable input table mapping if set
7	use slot 3 firmware for input if set

The screen flags are defined as follows:

Bit	Meaning
0	enable output table mapping if set
1	user slot 3 firmware for output if set
2	map linefeed to cr/lf if set
3	set high bit before outputting if set
4	use special reverse scroll routine if set
5-7	undefined

The printer flags are defined as follows:

Bit	Meaning
0	follow carriage return with linefeed if set
1	output a form feed when closed if set
2	set high bit before outputting if set

5.11.4 Adding a New Driver

To add a new driver, the source to the current drivers must be removed from the archive called "SHELLDEV.ARC". Then, compile and assemble the files contained in the archive. Edit the file "devtab.a65" to add the new driver name and the names of the routines used to access it. Then, use the "linkdev.sh" batch file as a model to link the driver together.

Once the driver is created, use the LDEV program to load the driver into Bank 1. The driver will be used automatically by the SHELL. Once the driver is working, the ".profile" file can be modified to load the driver as part of the booting process.

5.12   Debugging Pseudo-code

The purpose of this section is to provide information which will facilitate the debugging of C language routines which have been compiled with the Aztec C pseudo-code compiler CCI. This is not a tutorial on debugging techniques in general. The information and techniques provided pobably need only be used when embedding print statements fails to locate the problem.

The first piece of information that is absolutely necessary is the load map or namelist of the program. This is obtained by giving the -t option the the Aztec linker. The result will be a table of global names and the their addresses. The list is not sorted numerically. The list should be printed, or else the important addresses copied to paper.

The program should then be loaded into memory using either the SHELL load command or the DOS BLOAD command. From DOS, exit to the monitor by typing:

CALL -151

From the SHELL, simply type "bye".

The beginning of each C routine starts with a 6502 'jsr' instruction. This is a call to the pseudo-code interpreter. It is immediately followed by two bytes which specify the amount of stack space to reserve for local variables. Immediately following those two bytes are the pseudo-code opcodes.

For pseudo-code opcodes, the value FFH will cause the interpreter to jump to a brk instruction. The 6502 brk instruction will always be at the same address for the interpreter break-point instruction. The interpreter's program counter is stored in locations 6 and 7. It will contain the address of the breakpoint plus one.

To breakpoint at the beginning of a particular routine, simply find the address of the routine in the namelist, and replace the first byte (which will be a 20H) with a 00H. Then, if running under DOS, start the program at its load point. If running under the SHELL, warm boot the shell by typing:

3D0G

Then in response to the SHELL prompt, type:

run arg1 arg2 arg3

where the arguments need only be specified if required. When the breakpoint is encountered, control will revert to the monitor. At any breakpoint, any global variables, whose addresses may be found in the namelist, may be observerd or modified using the normal monitor commands.

At the beginning of a routine there is some additional information that is available. First, the address to which the current routine would have returned will still be in the intepreter's program counter (locations 6-7). If there were any arguments, the instruction at that address will be an AEH or AFH which are stack modification opcodes.

Secondly, the pseudo-stack pointer (locations 2-3) will be pointing at the first of any arguments which were passed to the current routine. Any additional arguments will follow in order.

There is also a frame pointer in locations 4-5. This pointer contains the address of the previous routine's stack frame which is diagrammed below.

             OLD PC
             OLD FRAME
             OLD SP
             RETURN ADDR   - FRAME POINTER

Each value is two bytes. The return address is the actual return address, which will ususally be the same value as it is somewhere within the interpreter itself. Immediately following the stack frame are the arguments which were passed to that routine.

Note that by using the old frame pointer, the entire calling sequence may be traced backwards. The arguments to each routine may also be examined. Local variables of a routine are found before its stack frame, arguments after.

The final method of interest involves determining the value returned from a routine. For the purposes of discussion, we shall assume that routine A calls routine B and we wish to determine the value returned by by B. The first thing that must be done, is to determine the return address of the routine B. This can be done in two ways. First, if the routine is called from only one point, set a breakpoint at the beginning of routine B. Then when the breakpoint is hit, look in locations 6-7 for the return address.

If routine B may be called from a number of places, we must use a different method to find the return address. We must look through the bytes in routine A for a call to routine B. This is possible since the opcode for a function call is ACH. It will immediately be followed by the address of routine B. The very next address is the return address from B.

Once the return address has been fround, replace the byte there with FFH and restart the program. When the breakpoint is encountered, the return value will be in locations 8-9 (8-B if a long is returned).

Note that continuing from these breakpoints is not possible

Appendices

APPENDIX A

Compiler Error Codes

No.   Interpretation

```
 1:  bad digit in octal constant
 2:  string space exhausted
 3:  unterminated string
 4:  internal error
 5:  illegal type for function
 6:  inappropriate arguments
 7:  bad declaration syntax
 8:  syntax error in typecast
 9:  array dimension must be constant
10:  array size must be positive integer
11:  data type too complex
12:  illegal pointer reference
13:  unimplemented type
14:  internal
15:  internal
16:  data type conflict
17:  internal
18:  data type conflict
19:  obsolete
20:  structure redeclaration
21:  missing }
22:  syntax error in structure declaration
23:  obsolete
24:  need right parenthesis or comma in arg list
25:  structure member name expected here
26:  must be structure/union member
27:  illegal typecast
28:  incompatible structures
29:  illegal use of structure
30:  missing : in ? conditional expression
31:  call of non-function
32:  illegal pointer calculation
33:  illegal type
34:  undefined symbol
35:  typedef not allowed here
36:  no more expression space
37:  invalid expression for unary operator
38:  no auto. aggregate initialization allowed
39:  obsolete
40:  internal
41:  initializer not a constant
42:  too many initializers
43:  initialization of undefined structure
44:  obsolete
45:  bad declaration syntax
46:  missing closing brace
47:  open failure on include file
48:  illegal symbol name
49:  multiply defined symbol
```

```
50:   missing bracket
51:   lvalue required
52:   obsolete
53:   multiply defined label
54:   too many labels
55:   missing quote
56:   missing apostrophe
57:   line too long
58:   illegal # encountered
59:   macro space overflow
60:   obsolete
61:   reference of member of undefined structure
62:   function body must be compound statement
63:   undefined label
64:   inappropriate arguments
65:   illegal argument name
66:   expected comma
67:   invalid else
68:   syntax error
69:   missing semicolon
70:   goto needs a label
71:   statement syntax error in do-while
72:   statement syntax error in for
73:   statement syntax error in for
74:   case value must integer constant
75:   missing colon on case
76:   too many cases in switch
77:   case outside of switch
78:   missing colon on default
79:   duplicate default
80:   default outside of switch
81:   break/continue error
82:   illegal character
83:   too many nested includes
84:   too many dimensions in array declaration
85:   not an argument
86:   null dimension in array
87:   invalid character constant
88:   not a structure
89:   invalid use of register storage class
90:   symbol redeclared
91:   illegal use of floating point type
92:   illegal type conversion
93:   illegal expression type for switch
94:   invalid identifier in macro definition
95:   macro needs argument list
96:   missing argument to macro
97:   obsolete
98:   not enough arguments in macro reference
99:   internal
100:  internal
101:  missing close parenthesis on macro reference
102:  macro arguments too long
103:  #else with no #if
104:  #endif with no #if
```

```
105:   #endasm with no #asm
106:   #asm within #asm block
107:   missing #endif
108:   missing #endasm
109:   #if value must be integer constant
110:   invalid use of : operator
111:   invalid use of void expression
112:   invalid use function pointer
113:   duplicate case in switch
114:   macro redefined
115:   keyword redefined
```

Explanations

1:   bad digit in octal constant

The only numerals permitted in the base 8 (octal) counting system are zero through seven. In order to distinguish between octal, hexadecimal, and decimal constants, octal constants are preceded by a zero. Any number beginning with a zero must not contain a digit greater than seven. Octal constants look like this: 01, 027, 003. Hexadecimal constants begin with 0x (e.g., 0x1, 0xAA0, 0xFFF).

2:   string space exhausted

The compiler maintains an internal table of the strings appearing in the source code. Since this table has a finite size, it may overflow during compilation and cause this error code. The table default size is about one or two thousand characters depending on the operating system. The size can be changed using the compiler option **-Z**. Through simple guesswork, it is possible to arrive at a table size sufficient for compiling your program. The following example illustrates the use of this option:

c65 -Z3000 bigexmpl.c

The new table size allows the strings in the file to total 3000 bytes in length. This is equal to 3000 characters.

3:   unterminated string

All strings must begin and end with double quotes ("). This message indicates that a double quote has remained unpaired.

4:   internal error

This error message should not occur. It is a check on the internal workings of the compiler and is not known to be caused by any particular piece of code. However, if this error code appears, please bring it to the attention of MANX. It could be a bug in the compiler. The release documentation enclosed with the product contains further information.

5: illegal type for function

The type of a function refers to the type of the value which it returns. Functions return an **int** by default unless they are declared otherwise. However, functions are not allowed to return aggregates (arrays or structures). An attempt to write a function such as **struct sam func()** will generate this error code. The legal function types are **char, int, float, double, unsigned, long, void** and a pointer to any type (including structures).

6:   error in argument declaration

The declaration list for the formal parameters of a function stands immediately before the left brace of the function body, as shown below. Undeclared arguments default to **int**, though it is usually better practice to declare everything. Naturally, this declaration list may be empty, whether or not the function takes any arguments at all.

No other inappropriate symbols should appear before the left (open) brace.

```
badfunction(arg1, arg2)
shrt arg 1;      /* misspelled or invalid keyword */
double arg 2;
{ /* function body */
}

goodfunction(arg1,arg2)
float arg1;
int arg2;    /* this line is not required */
{ /* function body */
}
```

7: bad declaration syntax

A common cause of this error is the absence of a semicolon at the end of a declaration. The compiler expects a semicolon to follow a variable declaration unless commas appear between variable names in multiple declarations.

```
int i, j;               /* correct */
char c d;                /* error 7 */
char *s1, *s2
float k;                /* error 7 detected here */
```

Sometimes the compiler may not detect the error until the next program line. A missing semicolon at the end of a #include'd file will be detected back in the file being compiled or in another #include file. This is a good example of why it is important to examine the context of the error rather than to rely solely on the information provided by the compiler error message(s).

8:   syntax error in type cast

The syntax of the cast operator must be carefully observed. A common error is to omit a parenthesis:

```
i = 3 * (int number);          /* incorrect usage */
i = 3 * ((int)number);         /* correct usage */
```

9:   array dimension must be constant

The dimension given an array must be a constant fo type **char, int, or unsigned**. This value is specified in the declaration of the array. See error 10.

10: array size must be positive integer

The dimension of an array is required to be greater than zero. A dimension less than or equal to zero becomes 1 by default. As can be seen from the following example, specifying a dimension of zero is not the same as leaving the brackets empty.

```
char badarray[0];       /* meaningless */
extern char goodarray[];        /* good */
```

Empty brackets are used when declaring an array that has been defined (given a size and storage in memory) somewhere else (that is, outside the current function or file). In the above example, **goodarray** is external. Function arguments should be declared with a null dimension:

```
func(s1,s2)
char s1[], s2[];
{
        ...
}
```

11:   data type too complex

This message is best explained by example:

char ******foo;

The form of this declaration implies five pointers-to-pointers. The sixth asterisk indicates a pointer to a **char**. The compiler is unable to keep track of so many "levels". Removing just one of the asterisks will cure the error; all that is being declared in any case is a single two-byte pointer. However it is to be hoped that such a construct will never be needed.

12:   illegal pointer reference

The type of a pointer must be either **int** or **unsigned**. This is why you might get away with not declaring pointer arguments and functions like **fopen** which return a pointer; they default to **int**. When this error is generated, an expression used as a pointer is of an invalid type:

```
          char c;
          int var;            /* any variable */
          int varaddress;
          varaddress = &var;      /* valid since addresses */
          *(varaddress) = 'c';     /* can fit in an int; */
          *(expression) = 10;    /* in general, expression
                          must be an int or unsigned */
          *c = 'c';      /* error 12 */
```

13:   internal   [see error 4]

14:   internal   [see error 4]

15:   storage class conflict

Only automatic variables and function parameters can be specified as **register**.
This error can be caused by declaring a **static register** variable. While structure members cannot be given a storage class at all, function arguments can be specified only as **register**.
A **register int i** declaration is not allowed outside a function--it will generate error 89 (see below).

16: data type conflict

The basic data types are not numerous, and there are not many ways to use them in declarations. The possibilities are listed below.

This error code indicates that two incompatible data types were used in conjunction with one another. For example, while it is valid to say **long int i**, and **unsigned int j**, it is meaningless to use **double int k** or **float char c**. In this respect, the compiler checks to make sure that **int**, **char**, **float** and **double** are used correctly.

data type	interpretation	size(bytes)
char	character	1
int	integer	2
unsigned/unsigned int	unsigned integer	2
short	integer	2
long/long integer	long integer	4
float	floating point number	4
long float/double	double precision float	8

17: internal error   [see error 4]

18: data type conflict

This message indicates an error in the use of the **long** or **unsigned** data type. **long** can be applied as a qualifier to **int** and **float**. **unsigned** can be used with **int.**

```
        long i;                          /* a long int */
        long float d;                    /* a double */
        unsigned u;                /* an unsigned int */
        unsigned char c;
        unsigned long l;
        unsigned float f;                /* error 18 */
```

19: obsolete

Error codes interpreted as obsolete do not occur in release 1.05 of the compiler. Some simply no longer apply due to the increased adaptibility of the compiler. Other error codes have been translated into full messages sent directly to the screen. If you are using an older version of the product and have need of these codes, please contact MANX for information.

20: structure redeclaration

The compiler is able to tell you if a structure has already

been defined. This message informs you that you have tried to redefine a **structure**.

21:· **missing }**

The compiler expects to find a comma after each member in the list of fields for a **structure** initialization. After the last field, it expects a right (close) brace.

```
struct sam {
      int bone;
      char license[10];
} harry = {
      1,
      123-4-1984;
```

22: **syntax error in structure declaration**

The compiler was unable to find the left (open) brace which follows the tag in a **structure** declaration. In the example for error 21, "sam" is the structure tag. A left brace must follow the keyword **struct** if no structure tag is specified.

23: **obsolete**   [see error 19]

24: **need right parenthesis or comma**

The right parenthesis is missing from a function call. Every function call must have an argument list enclosed by parentheses even if the list is empty. A right parenthesis is required to terminate the argument list.

In the following example, the parentheses indicate that **getchar** is a function rather than a variable.

```
getchar();
```

This is the equivalent of

```
CALL getchar
```

which might be found in a more explicit programming language. In general, a function is recognized as a name followed by a left parenthesis.

With the exception of reserved words, any name can be made a function by the addition of parentheses. However, if a previously defined variable is used as a function name, a compilation error will result.

Moreover, a comma must separate each argument in the list.

```
struct sam harry;
struct sam thom;
    ...
harry = thom;
```

29: illegal use of structure

Not all operators can accept a structure as an operand. Also, structures cannot be passed as arguments. However, it is possible to take the address of a structure using the ampersand (&), to assign structures, and to reference a member of a structure using the dot operator.

30: missing : in ? conditional expression

The standard syntax for this operator is

expression ? statement1 : statement2 .

It is not desirable to use ?: for extremely complicated expressions; its purpose lies in brevity and clarity.

31: call of non-function

The following is format for the call of a function:

symbol(arg1, arg2, ..., argn)

where "symbol" is not a reserved word and the expression stands in the body of a function. Error 31, in reference to the expression above, indicates that "symbol" has been previously declared as something other than a function.

A missing operator may also cause this error:

```
a(b + c);              /* error 31 */
a * (b + c);           /* intended */
```

The missing '*' makes the compiler view "a()" as a function call.

32: illegal pointer calculation

Pointers may be involved in three calculations. An integral value can be added to or subtracted from a pointer. Pointers to objects of the same type can be subtracted from one another and compared to one another. (For a formal definition, see Kernighan and Ritchie pp. 188-189.) Since the comparison and subtraction of two pointers is dependent upon pointer size, both operands must be the same size.

33: illegal type

The unary minus (-) and bit complement (~) operators cannot be applied to structures, pointers, arrays and functions. There is no reasonable interpretation for the following:

```
int function();
char array[12];
struct sam { ... } harry;
a = -array;                    /* ? */
b = -harry;
c = ~function & WRONG;
```

34: undefined symbol

The compiler will recognize only reserved words and names which have been previously defined. This error is often the result of a typographical error or due to an omitted declaration.

35: typedef not allowed here

Symbols which have been defined as types are not allowed within expressions. The exception to this rule is the use of **sizeof(expression)** and the cast operator. Compare the accompanying examples:

```
struct sam {
    int i;
    } harry;
typedef double bigfloat;
typedef struct sam foo;

j = 4 * bigfloat f;            /* error 35 */
k = &foo;                      /* error 35 */
x = sizeof(bigfloat);
y = sizeof(foo);                   /* good */
```

The compiler will detect two errors in this code. In the first assignment, a typecast was probably intended; compare error 8. The second assignment makes reference to the address of a structure type. However, the structure type is just a template for instances of the structure (such as "harry"). It is no more meaningful to take the address of a structure type than any other data type, as in &int

36: no more expression space

This message indicates that the expression table is not large enough for the compiler to process the source code. It is necessary to recompile the file using the -E option to increase the number of available entries in the expression table. See the description of the compiler in the manual.

The command sequence should look like this:

            c65 -E500 filename.c

37: invalid expression

This error occurs in the evaluation of an expression containing a unary operator. The operand either is not given or is itself an invalid expression.

Unary operators take just one operand; they work on just one variable or expression. If the operand is not simply missing, as in the example below, it fails to evaluate to anything its operator can accept. The unary operators are logical not (!), bit complement (~), increment (++), decrement (--), unary minus (-), typecast, pointer-to (*), address-of (&), and sizeof.

            if (!) ;

38: no auto. aggregate initialization

It is not permitted to initialize automatic arrays and structures. Static and external aggregates may be initialized, but by default their members are set to zero.

```
        char array[5] = { 'a', 'b', 'c', 'd' };
        function()
        {
            static struct sam {
                int bone;
                char license[10];
            } harry = {
                1,
                "123-4-1984"
            };
            char autoarray[2] = { 'f', 'g' };    /* no good */
            extern char array[];
        }
```

There are three variables in the above example, only two of which are correctly initialized. The variable "array" may be initialized because it is external. Its first four members will be given the characters as shown. The fifth member will be set to zero.
The structure "harry" is static and may be initialized. Notice that "license" cannot be initialized without first giving a value to "bone". There are no provisions in C for setting a value in the middle of an aggregate.
The variable "autoarray" is an automatic array. That is, it is local to a function and it is not declared to be static. Automatic variables reappear automatically every time a function is called, and they are guaranteed to contain garbage. Automatic

aggregates cannot be initialized.

39:   obsolete   [see error 19]

40:   internal   [see error 4]

41:   initializer not a constant

In certain initializations, the expression to the right of the equals sign (=) must be a constant. Indeed, only automatic and register variables may be initialized to an expression. Such initializations are meant as a convenient shorthand to eliminate assignment statements. The initialization of statics and globals actiually occurs at link-time, and not at run-time.

```
{
int i = 3;
static int j = (2 + i);          /* illegal */
}
```

42:   too many initializers

There were more values found in an initialization than array or structure members exist to hold them. Either too many values were specified or there should have been more members declared in the aggregate definition.

In the initialization of a complex data structure, it is possible to enclose the initializer in a single set of braces and simply list the members, separated by commas. If more than one set of braces is used, as in the case of a structure within a structure, the initializer must be entirely braced.

```
struct {
     struct {
          char array[];
     } substruct;
} superstruct =
```

version 1:
```
{
    "abcdefghij"
};
```
version 2:
```
{
    {
      { 'a','b','c',...,'i','j'}
    }
};
```

In version 1, the initializers are copied byte-for-byte onto

the structure, **superstruct**.

Another likely source of this error is in the initialization of arrays with strings, as in:

    char array[10]= "abcdefghij";

This will generate error 42 because the string constant on the right is null-terminated. The null terminator ('\0' or 0x00) brings the size of the initializer to 11 bytes, which overflows the ten-byte array.

43:   undefined structure initialization

An attempt has been made to assign values to a structure which has not yet been defined.

    struct sam {...};
    struct dog sam = { 1, 2, 3};   /* error 43 */

44:   obsolete   [see error 19]

45:   bad declaration syntax

This error code is an all purpose means for catching errors in declaration statements. It indicates that the compiler is unable to interpret a word in an external declaration list.

46:   missing closing brace

All the braces did not pair up at the end of compilation. If all the preceding code is correct, this message indicates that the final closing brace to a function is missing. However, it can also result from a brace missing from an inner block.

Keep in mind that the compiler accepts or rejects code on the basis of syntax, so that an error is detected only when the rules of grammar are violated. This can be misleading. For example, the program below will generate error 46 at the end even though the human error probably occurred in the **while** loop several lines earlier.

As the code appears here, every statement after the left brace in line 6 belongs to the body of the **while** loop. The compilation error vanishes when a right brace is appended to the end of the program, but the results during run time will be indecipherable because the brace should be placed at the end of the loop.

It is usually best to match braces visually before running the compiler. A C-oriented text editor makes this task easier.

```
            main()
            {
                int i, j;
                char array[80];

                gets(array);
                i = 0;
                while (array[i])  {
                    putchar(array[i]);
                    i++;
                for ( i=0; array[i];i++) {
                    for (j=i + 1; array[j]; j++)  {
                        printf("elements %d and %d are ", i, j);
                        if (array[i] == array[j])
                            printf("the same\n");
                        else printf("different\n");
                    }
                    putchar('\n');
                }
            }
```

47: open failure on include file

When a file is #included, the compiler will look for it in a
default area (see the manual description of the compiler). This
message will be generated if the file could not be opened. An
open failure usually occurs when the included file does not exist
where the compiler is searching for it. Note that a drive
specification is allowed in an include statement, but this
diminishes flexibility somewhat.

48: illegal symbol name

This message is produced by the preprocessor, which is that
part of the compiler which handles lines which begin with a pound
sign (#). The source for the error is on such a line. A legal
name is a string whose first character is an alphabetic (a letter
of the alphabet or an underscore). The succeeding characters may
be any combination of alphanumerics (alphabetics and numerals).
The following symbols will produce this error code:

            2nd_time,
            dont_do_this!

49: multiply defined symbol

This message warns that a symbol has already been declared
and that it is illegal to redeclare it. The following is a
representative example:

            int i, j, k, i;   /* illegal */

50: missing bracket

This error code is used to indicate the need for a parenthesis, bracket or brace in a variety of circumstances.

51: lvalue required

Only **lvalues** are are allowed to stand on the left-hand side of an assignment. For example:

```
        int num;
        num = 7;
```

They are distinguished from **rvalues**, which can never stand on the left of an assignment, by the fact that they refer to a unique location in memory where a value can be stored. An **lvalue** may be thought of as a bucket into which an **rvalue** can be dropped. Just as the contents of one bucket can be passed to another, so can an lvalue **y** be assigned to another lvalue, **x**:

```
        #define NUMBER    512
        x = y;
        1024 = z;      /* wrong; l/rvalues are reversed */
        NUMBER = x;    /* wrong; NUMBER is still an rvalue */
```

Some operators which require **lvalues** as operands are increment (++), decrement (--), and address-of (&). It is not possible to take the address of a register variable as was attempted in the following example:

```
        register int i, j;
        i = 3;
        j = &i;
```


52: obsolete   [see error 16]

53: multiply defined label

On occasions when the goto statement is used, it is important that the specified label be unique. There is no criterion by which the computer can choose between identical labels. If you have trouble finding the duplicate label, use your text editor to search for all occurrences of the string.

54: too many labels

The compiler maintains an internal table of labels which will support up to several dozen labels. Although this table is

fixed in size, it should satisfy the requirements of any
reasonable C program.   C was structured to discourage
extravagance in the use of goto's.   Strictly speaking, goto
statements are not required by any procedure in C; they are
primarily recommended as a quick and simple means of exiting from
a nested structure.

This error indicates that you should significantly reduce
the number of goto's in your program.

55: missing quote

The compiler found a mismatched double quote (") in a
#define preprocessor command. Unlike brackets, quotes are not
paired innermost to outermost, but sequentially. So the first
quote is associated with the second, the third with the fourth,
and so on. Single quotes (') and double quotes (") are entirely
different characters and should not be confused. The latter are
used to delimit string constants. A double quote can be included
in a string by use of a backslash, as in this example:

                    "this is a string"
                    "this is a string with an embedded quote: \". "

56: missing apostrophe

The compiler found a mismatched single quote or apostrophe
(') in a #define preprocessor command. Single quotes are paired
sequentially (see error 55). Although quotes can not be nested,
a quote can be represented in a character constant with a
backslash:

            char c = '\'';   /* c is initialized to
                                     single quote */

57: line too long

Lines are restricted in length by the size of the buffer
used to hold them. This restriction varies from system to
system. However, logical lines can be infinitely long by
continuing a line with a backslash-newline sequence. These
characters will be ignored.

58: illegal # encountered

The pound sign (#) begins each command for the preprocessor:
#include, #define, #ifdef, #ifndef, #else, #endif, #asm, #endasm,
#line and #undef.   These symbols are strictly defined. The
pound sign (#) must be in column one and lower case letters are
required.

59: macro table overflow

Macros can be defined with a preprocessor command of the following form:

#define [identifier] [substitution text]

The compiler then proceeds to replace all instances of "identifier" with the substitution text that was specified by the #define.

This error code means that the table in which macros are stored has overflowed. The -X compiler option can be used to specify a larger macro table.

60: obsolete   [see error 19]

61: reference of member of undefined structure

Occurs only under compilation without the -S option. Consider the following example:

```
int bone;
struct cat {
    int toy;
} manx;
struct dog *samptr;
manx.toy = 1;
bone = samptr->toy;         /* error 61 */
```

This error code appears most often in conjunction with this kind of mistake. It is possible to define a pointer to a structure without having already defined the structure itself. In the example, **samptr** is a structure pointer, but what form that structure ("dog") may take is still unknown. So when reference is made to a member of the structure to which **samptr** points, the compiler replies that it does not even known what the structure looks like.

The **-S** compiler option is provided to duplicate the manner in which earlier versions of UNIX treated structures. Given the example above, it would make the compiler search all previously defined structures for the member in question. In particular, the value of the member "toy" found in the structure "manx" would be assigned to the variable "bone". The -S option is not recommended as a short cut for defining structures.

62: function body must be compound statement

The body of a function must be enclosed by braces, even though it may consist of only one statement:

```
function()
{
```

```
                        return 1;
            }
```

63: undefined label

A **goto** statement is meaningless if the corresponding label does not appear somewhere in the code. The compiler disallows this since it must be able to specify a destination to the computer.

It is not possible to goto a label outside the present function (labels are local to the function in which they appear). Thus, if a label does not exist in the same procedure as its corresponding goto, this message will be generated.

64: inappropriate arguments

When a function is declared (as opposed to defined), it is poor syntax to specify an argument list:

```
            function(string)
            char *string;
            {
                char *func1();          /* correct */
                double func2(x,y);      /* wrong */
                    ...
            }
```

In this example, function() is being defined, but func1() and func2() are being declared.

65: illegal or missing argument name

The compiler has found an illegal name in a function argument list. An argument name must conform to the same rules as variable names, beginning with an alphabetic (letter or underscore) and continuing with any sequence of alphanumerics and underscores. Names must not coincide with reserved words.

66: expected comma

In an argument list, arguments must be separated by commas.

67: invalid else

An **else** was found which is not associated with an **if** statement. **else** is bound to the nearest **if** at its own level of nesting. So if-else pairings are determined by their relative placement in the code and their grouping by braces.

```
            if(...)  {
```

```
        ...
        if (...) {
            ...
        } else if (...)
            ...
        } else {
            ...
        }
```

The indentation of the source text should indicate the intended structure of the code. Note that the indentation of the if and else-if means only that the programmer wanted both conditionals to be nested at the same level, in particular one step down from the presiding if statement. But it is the placement of braces that determines this for the compiler. The example above is correct, but probably does not conform to the expectations revealed by the indentation of the else statement. As shown here, the else is paired with the first if, not the second.

68: syntax error

The keywords used in declaring a variable, which specify storage class and data type, must not appear in an executable statement. In particular, all local declarations must appear at the beginning of a block, that is, directly following the left brace which delimits the body of a loop, conditional or function. Once the compiler has reached a non-declaration, a keyword such as char or int must not lead a statement; compare the use of the casting operator:

```
func()
{
        int i;
        char array[12];
        float k = 2.03;

        i = 0;
        int m;                    /* error 68 */
        j = i + 5;
        i = (int)k;               /* correct */
        if (i) {
                int i = 3;
                j = i;
                printf("%d",i);
        }
        printf("%d%d\n",i,j);
}
```

This trivial function prints the values 3, 2 and 3. The variable i which is declared in the body of the conditional (if) lives only until the next right brace; then it dies, and the original i regains its identity.

69: missing semicolon

A semicolon is missing from the end of an executable statement. This error code is subject to the same vagaries as its cousin, error 7. It will remain undetected until the following line and is often spuriously caused by a previous error.

70: bad goto syntax

Compare your use of goto with an example. This message says that you did not specify where you wanted to goto with a label:

```
                    goto label;
                        ...
            label:
                        ...
```

It is not possible to goto just any identifier in the source code; labels are special because they are followed by a colon.

71:   statement syntax error in do-while

The body of a **do-while** may consist of one statement or several statements enclosed in braces. A **while** conditional is required after the body of the loop. This is true even if the loop is infinite, as it is required by the rules of syntax. After typing in a long body, don't forget the **while** conditional.

72:   statement syntax error in for

This error focuses on another control flow statement, the **for**. The keyword, **for**, must be followed by parentheses. In the parentheses belong three expressions, any or all of which may be null. For the sake of clarity, C requires that the two semi-colons which separate the expressions be retained, even if all three expressions are empty.

```
        for (;;)        /* an infinite loop which does */
        ;               /* absolutely nothing */
```

73:   statement syntax error in for

See error 72.

74: case value must be integer constant

Strictly speaking, each value in a **case** statement must be a constant of one of three types: **char**, **int** or **unsigned**. This is similar to the rule for a **switched** variable. In the following example, a float must be cast to an int in order to be switched;

79: duplicate default

The compiler has found more than one **default** in a **switch**. Switch will compare a variable to a given list of values. But it is not always possible to anticipate the full range of values which the variable may take. Nor is it feasible to specify a large number of cases in which the program is not particularly interested.

So C provides for a default case. The default will handle all those values not specified by a case statement. It is analogous to the else companion to the conditional, if. Just as there is one else for every if, only one default case is allowed in a switch statement. However, unlike the else statement, the position of a default is not crucial; a default can appear anywhere in a list of cases.

80: default outside of switch

The keyword, "default", is used just like "case". It must appear within the brackets which delimit the switch statement.

81: break/continue error

Break and continue are used to skip the remainder of a loop in order to exit or repeat the loop. Break will also end a switch statement. But when the keywords, **break** or **continue**, are used outside of these contexts, this message results.

82: illegal character

Some characters simply do not make sense in a C program, such as '$' and '@'. Others, for instance the pound sign (#), may be valid only in particular contexts.

83: too many nested includes

#includes can be nested, but this capacity is limited. The compiler will balk if required to descend more than three levels into a nest. In the example given, file D is not allowed to have a #include in the compilation of file A.

```
    file A          file B          file C          file D
 #include "B"    #include "C"    #include "D"
```

84: too many dimensions for array

An array can have a maximum of five dimensions; if more are specified when the array is declared, this message occurs.

85: not an argument

The compiler has found a name in the declaration list that was not in the argument list. Only the converse case is valid, i.e., an argument can be passed and not subsequently declared.

86: null dimension in array

In certain cases, the compiler knows how to treat multidimensional arrays whose left-most dimensions are not given in its declaration. Specifically, this is true for an extern declaration and an array initialization. The value of any dimension which is not the left-most must be given.

```
        extern char array[][12];        /* correct */
        extern char badarray[5][];      /* wrong */
```

87: invalid character constant

Character constants may consist of one or two characters enclosed in single quotes, as 'a' or 'ab'. There is no analog to a null string, so '' (two single quotes with no intervening white space) is not allowed. Recall that the special backslash characters (\b, \n, \t etc.) are singular, so that the following are valid: '\n', '\na', 'a\n'; 'aaa' is invalid.

88: not a structure

Occurs only under compilation without the -S option. A name used as a structure does not refer to a structure, but to some other data type.

```
        int i;
        i.member = 3;            /* error 88 */
```

89: invalid storage class

A globally defined variable cannot be specified as register. Register variables are required to be local.

90: symbol redeclared

A function argument has been declared more than once.

91: illegal use of floating point type

Floating point numbers can be negated (unary minus), added, subtracted, multiplied, divided and compared; any other operator

will produce this error message.

92: illegal type conversion

This error code indicates that a data type conversion, implicit in the code, is not allowed, as in the following piece of code:

```
int i;
float j;
char *ptr;
...
i = j + ptr;
```

The diagram shows how variables are converted to different types in the evaluation of expressions. Initially, variables of type **char** and **short** become **int**, and **float** becomes **double**. Then all variables are promoted to the highest type present in the expression. The result of the expression will have this type also. Thus, an expression containing a **float** will evaluate to a **double**.

hierarchy of types:

```
double <-- float
long
unsigned
int <-- short, char
```

93: illegal expression type for switch

Only a **char**, **int** or **unsigned** variable can be switched. See the example for error 74.

94: bad argument to define

An illegal name was used for an argument in the definition of a macro. For a description of legal names, see error 65.

95: no argument list

When a macro is defined with arguments, any invocation of that macro is expected to have arguments of corresponding form. This error code is generated when no parenthesized argument list was found in a macro reference.

```
#define   getchar()   getc(stdin)
   ...
   c = getchar;          /* error 95 */
```


96: missing argument to macro

Not enough arguments were found in an invocation of a macro. Specifically, a "double comma" will produce this error:

> #define reverse(x,y,z)   (z,y,x)
>
> func(reverse(i,,k));

97: obsolete   [see error 19]

98: not enough args in macro reference

The incorrect number of arguments was found in an invocation of a previously defined macro. As the examples show, this error is not identical to error 96.

> #define   exchange(x,y)   (y,x)
>
> func(exchange(i));   /* error 98 */

99: internal   [see error 4]

100: internal   [see error 4]

101: missing close parenthesis on macro reference

A right (close) parenthesis is expected in a macro reference with arguments. In a sense, this is the complement of error 95; a macro argument list is checked for both a beginning and an ending.

102: macro arguments too long

The combined length of a macro's arguments is limited. This error can be resolved by simply shortening the arguments with which the macro is invoked.

103: #else with no #if

Correspondence between #if and #else is analogous to that which exists between the control flow statements, if and else. Obviously, much depends upon the relative placement of the statements in the code. However, #if blocks must always be terminated by #endif, and the #else statement must be included in the block of the #if with which it is associated. For example:

> #if ERROR > 0


```
                        printf("there was an error\n");
        #else
                        printf("no error this time\n");
        #endif
```

#if statements can be nested, as below. The range of each #if is
determined by a #endif. This also excludes #else from #if blocks
to which it does not belong:

```
        #ifdef    JAN1
                  printf("happy new year!\n");
        #if       sick
                  printf("i think i'll go home now\n");
        #else
                  printf("i think i'll have another\n");
        #endif
        #else
                  printf("i wonder what day it is\n");
        #endif
```

If the first #endif was missing, error 103 would result. And
without the second #endif, the compiler would generate error 107.

104: #endif with no #if

    #endif is paired with the nearest #if, #ifdef or #ifndef
which precedes it. (See error 103.)

105: #endasm with no #asm

    #endasm must appear after an associated #asm. These
compiler-control lines are used to begin and end embedded
assembly code. This error code indicates that the compiler has
reached a #endasm without having found a previous #asm. If the
#asm was simply missing, the error list should begin with the
assembly code (which are undefined symbols to the compiler).

106: #asm within #asm block

    There is no meaningful sense in which in-line assembly code
can be nested, so the #asm keyword must not appear between a
paired #asm/#endasm. When a piece of in-line assembly is
augmented for temporary purposes, the old #asm and #endasm can be
enclosed in comments as place-holders.

```
        #asm
                /* temporary asm code */
        /*    #asm            old beginning   */
                /* more asm code */
        #endasm
```

107: missing #endif

A #endif is required for every #if, #ifdef and #ifndef, even if the entire source file is subject to a single conditional compilation. Try to assign pairs beginning with the first #endif. Backtrack to the previous #if and form the pair. Assign the next #endif with the nearest unpaired #if. When this process becomes greatly complicated, you might consider rethinking the logic of your program.

108: missing #endasm

In-line assembly code must be terminated by a #endasm in all cases. #asm must always be paired with a #endasm.

109: #if value must be integer constant

#if requires an integral constant expression. This allows both integer and character constants, the arithmetic operators, bitwise operators, the unary minus (-) and bit complement, and comparison tests.

Assuming all the macro constants (in capitals) are integers,

```
#if DIFF >= 'A'-'a'
#if (WORD &= ~MASK) >> 8
#if MAR | APR | MAY
```

are all legal expressions for use with #if.

110: invalid use of colon operator

The colon operator occurs in two places: 1. following a question mark as part of a conditional, as in (flag ? 1 : 0); 2. following a label inserted by the programmer or following one of the reserved labels, case and default.

111: illegal use of a void expression

This error can be caused by assigning a void expression to a variable, as in this example:

```
void func();
int h;

h = func(arg);
```

112: illegal use of function pointer

For example,

```
int (*funcptr) ();
    ...
funcptr++;
```

funcptr is a pointer to a function which returns an integer. Although it is like other pointers in that it contains the address of its object, it is not suject to the rules of pointer arithmetic. Otherwise, the offending statement in the example would be interpreted as adding to the pointer the size of the function, which is not a defined value.

113: duplicate case in switch

This simply means that, in a **switch** statement, there are two **case** values which are the same. Either the two **cases** must be combined into one, or one of them must be discarded. For instance:

```
switch (c) {
        case NOOP:
                return (0);
        case MULT:
                return (x * y);
        case DIV:
                return (x / y);
        case ADD:
                return (x + y);
        case NOOP:
        default:
                return;
}
```

The case of NOOP is duplicated, and will generate an error.

114: macro redefined

For example,

```
#define    islow(n)   (n>=0&&n<5)
    ...
#define    islow(n)   (n>=0&&n<=5)
```

The macro, **islow**, is being used to classify a numerical value. When a second definition of it is found, the compiler will compare the new substitution string with the previous one. If they are found to be different, the second definition will become current, and this error code will be produced.

In the example, the second definition differs from the first in a single character, '='. The second definition is also different from this one:

```
#define    islow(n)   n>=0&&n<=5
```


since the parentheses are missing.

The following lines will not generate this error:

```
#define    NULL 0
      ...
#define    NULL 0
```

But these are different from:

```
#define    NULL '\0'
```

In practice, this error message does not affect the compilation of the source code. The most recent "revision" of the substitution string is used for the macro. But relying upon this fact may not be a wise habit.

115: keyword redefined

Keywords cannot be defined as macros, as in:

```
#define    int  foo
```

If you have a variable which may be either, for instance, a short or a long integer, there are alternative methods for switching between the two. If you want to compile the variable as either type of integer, consider the following:

```
#ifdef    LONGINT
     long i;
#else
     short i;
#endif
```

Another possibility is through a **typedef**:

```
#ifdef    LONGINT
     typedef    long       VARTYPE;
#else
     typedef    short      VARTYPE;
#endif

VARTYPE i;
```

APPENDIX B

APPENDIX B - PROGRAMMING EXAMPLE

The following is a listing of the source and various intermediate
files produced from the following sequence of commands:

```
c65 -t -a prog.c
as65 -c -l prog.asm
ln -t prog.rel sh65.lib
```

The program listed shows how the same routine can be coded using
different "C" features to produce code that executes more
efficiently.

C SOURCE


```
main()
{
/*    The following examples show some of the ways that static,
      pointer, and register variables are used to produce faster
      executing code.
*/

/* good */
{
    int i, a[10];
    for (i=0;i<10;i++)
        a[i] = 2;
}

/* better */
{
    register i;
    static int a[10];
    for (i=0;i<10;i++)
        a[i] = 2;
}

/* best */
{
    register int *ip;
    static int a[10];

    for (ip=a; ip<a+10; )
        *ip++ = 2;
}
}
```

```
                    txa
                    adc       6+1
                    sta       25
                    clc
                    lda       28
                    adc       24
                    sta       16
                    lda       29
                    adc       25
                    sta       17
                    lda       #2
                    sta       (16,X)
                    txa
                    sta       (16),Y
                    jmp       .4
*}
*/* better */
*{
*       register i;
*       static int a[10];
                    dseg
                    rmb       20
                    cseg
*       for (i=0;i<10;i++)
                    stx       128
                    stx       129
                    jmp       .9
                    inc       128
                    bne       *+4
                    inc       129
                    lda       128
                    cmp       #10
                    lda       129
                    sbc       #0
                    bvs       *+7
                    bmi       *+5
                    jmp       .10
*         a[i] = 2;
                    lda       128
                    sta       28
                    lda       129
                    sta       29
                    lda       #1
                    ldy       #28
                    jsr       .shl#
                    clc
                    lda       28
                    adc       #<.7
                    sta       24
                    lda       29
                    adc       #>.7
                    sta       25
                    lda       #2
                    sta       (24,X)
                    txa
```

```
          sta       (24),Y
          jmp       .8
*}
*/* best */
*{
*     register int *ip;
*     static int a[10];
          dseg
          rmb       20
          cseg
*     for (ip=a; ip<a+10; )
          lda       #<.11
          sta       128
          lda       #>.11
          sta       129
          lda       128
          cmp       #<.11+20
          lda       129
          sbc       #>.11+20
          bcc       *+5
          jmp       .13
*       *ip++ = 2;
          clc
          lda       128
          sta       28
          adc       #2
          sta       128
          lda       129
          sta       29
          adc       #0
          sta       129
          lda       #2
          sta       (28,X)
          txa
          ldy       #1
          sta       (28),Y
          jmp       .12
*}
*}

          rts
          END
```

ASSEMBLER LISTING

```
 1 0000:                      *main()
 2 0000:                      *{
 3 0000:                             public  main_
 4 0000:   20 xx xx   main_    jsr     .csav#
 5 0003:   00                  fcb     .3
 6 0004:   EA FF               fdb     .2
 7 0006:                      */*  The following examples show some
 8 0006:                      *    pointer, and register variables
 9 0006:                      *    executing code.
10 0006:                      **/
11 0006:                      */*  good */
12 0006:                      *{
13 0006:                      *    int i, a[10];
14 0006:                      *    for (i=0;i<10;i++)
15 0006:   A0 FE               ldy     #254
16 0008:   91 06               sta     (6),Y
17 000a:   C8                  iny
18 000b:   91 06               sta     (6),Y
19 000d:   4C xx xx            jmp     .5
20 0010:                 .4    clc
21 0010:   18                  clc
22 0011:   A0 FE               ldy     #254
23 0013:   B1 06               lda     (6),Y
24 0015:   69 01               adc     #1
25 0017:   91 06               sta     (6),Y
26 0019:   C8                  iny
27 001a:   B1 06               lda     (6),Y
28 001c:   69 00               adc     #0
29 001e:   91 06               sta     (6),Y
30 0020:                 .5    ldy     #254
31 0020:   A0 FE               ldy     #254
32 0022:   B1 06               lda     (6),Y
33 0024:   C9 0A               cmp     #10
34 0026:   C8                  iny
35 0027:   B1 06               lda     (6),Y
36 0029:   E9 00               sbc     #0
37 002b:   70 05               bvs     *+7
38 002d:   30 03               bmi     *+5
39 002f:   4C xx xx            jmp     .6
40 0032:                 *        a[i] = 2;
41 0032:   88                  dey
42 0033:   B1 06               lda     (6),Y
43 0035:   85 1C               sta     28
44 0037:   C8                  iny
45 0038:   B1 06               lda     (6),Y
46 003a:   85 1D               sta     29
47 003c:   A9 01               lda     #1
48 003e:   A0 1C               ldy     #28
49 0040:   20 xx xx            jsr     .shl#
50 0043:   18                  clc
51 0044:   A9 EA               lda     #234
```

```
107 0098:    A5 1D            lda      29
108 009a:    69 xx            adc      #>.7
109 009c:    85 19            sta      25
110 009e:    A9 02            lda      #2
111 00a0:    81 18            sta      (24,X)
112 00a2:    8A               txa
113 00a3:    91 18            sta      (24),Y
114 00a5:    4C xx xx         jmp      .8
115 00a8:                .10
116 00a8:                *}
117 00a8:                */* best */
118 00a8:                *{
119 00a8:                *      register int *ip;
120 00a8:                *      static int a[10];
121 00a8:                       dseg
122 0014:                .11
123 0014:                       rmb      20
124 0028:                       cseg
125 00a8:                *      for (ip=a; ip<a+10; )
126 00a8:    A9 xx            lda      #<.11
127 00aa:    85 80            sta      128
128 00ac:    A9 xx            lda      #>.11
129 00ae:    85 81            sta      129
130 00b0:                .12
131 00b0:    A5 80            lda      128
132 00b2:    C9 xx            cmp      #<.11+20
133 00b4:    A5 81            lda      129
134 00b6:    E9 xx            sbc      #>.11+20
135 00b8:    90 03            bcc      *+5
136 00ba:    4C xx xx         jmp      .13
137 00bd:                *        *ip++ = 2;
138 00bd:    18               clc
139 00be:    A5 80            lda      128
140 00c0:    85 1C            sta      28
141 00c2:    69 02            adc      #2
142 00c4:    85 80            sta      128
143 00c6:    A5 81            lda      129
144 00c8:    85 1D            sta      29
145 00ca:    69 00            adc      #0
146 00cc:    85 81            sta      129
147 00ce:    A9 02            lda      #2
148 00d0:    81 1C            sta      (28,X)
149 00d2:    8A               txa
150 00d3:    A0 01            ldy      #1
151 00d5:    91 1C            sta      (28),Y
152 00d7:    4C xx xx         jmp      .12
153 00da:                .13
154 00da:                *}
155 00da:                *}
156 00da:    60               rts
157 00db:                .2       equ      -22
158 00db:                .3       equ      0
159 00db:                       END
```

SYMBOL TABLE FROM LN

```
0984 .asr
08DE .begin
0AEC .cali
09C3 .cpystk
0AC7 .cpystk2
0AA0 .cret
0A28 .csav
08DE .lngused
0A02 .modstk
0967 .shl
09A6 .shr
0B1E MEMRY
0B1A _Free_
0B18 _Top_
0B1C _closall
08F9 _croot_
0962 _exit_
0933 closall_
093A exit_
0803 main_
```